Inițiere în programarea orientată pe obiecte în limbajul C++

Adrian Runceanu

Inițiere în programarea orientată pe obiecte în limbajul C++

Copyright Adrian Runceanu

Publicata de Adrian Runceanu la Editura Digitala

Ediţie digitală publicată la Editura Digitală

Toate drepturile asupra ediţiei digitale a acestei cărţi sunt rezervate Editura Digitala. Nici o parte din această carte nu poate fi reprodusă sub nici o formă sau metodă fără acordul scris al autorului.

Aveţi drepturi de utiliza acestă carte doar pentru propria persoană. Această carte digitală nu poate fi revândută sau dată către alte persoane. Dacă doriţi să împărţiţi această carte cu o altă persoană, vă rugăm să achiziţionaţi încă o copie pentru fiecare cititor. Dacă citeşti această carte şi nu ai achiziţionat-o, te rugăm să accesezi magazinul preferat de cărţi digitale şi să-ţi cumperi propria copie.

Cuprins

Prolog
Capitolul 1
Capitolul 2
Capitolul 3
Capitolul 4
Capitolul 5
Capitolul 6
Capitolul 7
Capitolul 8
Capitolul 9
Capitolul 10
Bibliografie
Despre autor
Tine legatura cu Adrian Runceanu

Prolog

Scopul acestei carti este să prezinte într-o manieră accesibilă studentilor transpunerea conceptelor avansate ale orientării pe obiecte în limbajul de programare C++. Se prezintă conceptele de moştenire, clase derivate, clase virtuale şi proprietatea de polimorfism.

In conceperea acestei carti s-au avut în vedere in primul rand aspectele teoretice, care sunt prezentate detaliat, dar şi, aspectele practice, care sunt implementate in limbajul C++. Toate programele sursă prezentate în acest îndrumar sunt realizate în *limbajul de programare Borland C++* şi sunt verificate pe date de test variate, autorul garantând buna funcţionare a lor. Cartea se doreste a fi un ghid de programare orientata pe obiecte, care deschide interesul elevilor și studenţilor, problematica prezentată putând fi largită şi completată în cadrul unei apariţii ulterioare.

Scurt istoric C++

De la Wikipedia, enciclopedia liberă

C++ (pronunţat "si plus plus") este un limbaj de programare general. Este un limbaj multi-paradigmă cu verificarea statică a tipului variabilelor ce suportă programare procedurală, abstractizare a datelor, programare orientată pe obiect. În anii ’90, C++ a devenit unul din cele mai populare limbaje de programare comerciale. Nimeni nu deţine C++ însă, acesta fiind gratuit. Bjarne Stroustrup de la Bell Labs a dezvoltat C++ (iniţial denumit „C cu Clase”) în anii 1980 ca o serie de îmbunătăţiri ale limbajului C. Acestea au început cu adăugarea de clase apoi de funcţii virtuale, supraîncărcarea operatorilor, moştenire multiplă, template-uri şi excepţii. Limbajul de programare C++ a fost standardizat în 1998 ca şi ISO 14882:1998, versiunea curenta fiind din 2003, ISO 14882:2003.

Stroustrup a început să lucreze la C cu Clase în 1979. Ideea creării unui nou limbaj a venit din experienţa de programare pentru pregătirea tezei sale

de doctorat. Stroustrup a descoperit că Simula avea facilităţi foarte utile pentru proiecte mari, însă era prea lent, în timp ce BCPL era rapid, însă nu era de nivel înalt şi era nepotrivit pentru proiecte mari. Când Stroustrup a început să lucreze pentru Bell Labs, avea sarcina de a analiza nucleul UNIX referitor la calcul distribuit. Amintindu-şi de experienţa sa din perioada lucrării de doctorat, Stroustrup a început să îmbunătăţească C cu facilităţi asemănătoare Simula. C a fost ales deoarece era rapid şi portabil. La început facilităţile adaugate C-ului au fost clase, clase derivate, verificare a tipului, inline şi argumente cu valori implicite. În timp ce Stroustrup a proiectat C cu Clase (mai apoi C++), a scris de asemenea şi Cfront, un compilator care genera cod sursă C din cod C cu Clase. Prima lansare comercială a fost în 1985. În 1982, numele limbajului a fost schimbat de la C cu Clase la C++. Au fost adaugate noi facilităţi, inclusiv funcţii virtuale, supraîncărcarea operatorilor şi a functiilor, referinţe, constante, alocare dinamică, un control al tipului mai puternic şi noua variantă de comentariu pe un singur rând (liniile care încep cu caracterele '//').

În 1985 a fost lansată prima ediţie a cărţii "The C++ Programming Language" (Limbajul de programare C++), oferind informaţii importante despre limbaj, care încă nu era un standard oficial. În 1989 a fost lansată versiunea 2.0 a C++. Au apărut acum moştenirea multiplă, clase abstracte, funcţii statice, funcţii constante şi membri protected. În 1990 o altă carte a fost lansată, oferind suport pentru standarde viitoare. Ultimele adăugări includeau template-uri, excepţii, spaţii de nume (namespace-uri) şi tipul Boolean.

O dată cu evoluţia limbajului C++, a evoluat şi o bibliotecă standard. Prima adăugire a fost biblioteca de intrări/ieşiri (I/O stream), care oferea facilităţi pentru a înlocui funcţiile tradiţionale C cum ar fi printf şi scanf. Mai târziu, printre cele mai semnificative adăugari la biblioteca standard a fost STL (Standard Template Library) (Biblioteca de formate standard).

Capitolul 1 - Introducere în Programarea Orientată pe Obiecte

1. Programare procedurală – Programare orientată pe obiecte.

Limbajul C++, ca şi Pascal, utilizează modelul *programării structurate procedurale*, care constă în *descompunerea programului în proceduri* (funcţii), apelate în ordinea în care se desfăşoară algoritmul. Datele sunt *separate* de funcţiile care le manipulează.

Odată cu creşterea dimensiunii programelor s-a acordat o atenţie sporită organizării datelor – *funcţiile*, împreună cu *datele* pe care le manevrează sunt organizate ca un *modul*. *Programarea modulară* este tot programare procedurală, cu *proceduri şi date grupate în module* şi *ascunse* altor module.

Programarea orientată pe obiecte constă în *identificarea unor obiecte*, cu operaţii (metode) specifice asociate şi realizarea *comunicării* între aceste obiecte prin intermediul unor *mesaje*. Elementul constructiv – *obiectul* este o *instanţă a unei clase* (*tip de dată definită de utilizator*). Clasele sunt membre ale unei *ierarhii* şi sunt corelate între ele prin relaţii de *moştenire*.

Un limbaj de programare pune la dispoziţia utilizatorilor un număr de tipuri primitive (sau predefinite). Astfel în C++ avem ca tipuri primitive **char, int, float**, etc. Un tip de date este precizat printr-o mulţime finită de valori **T** (*constantele tipului*) şi o mulţime de operatori (aplicaţii $\mathbf{T} \rightarrow \mathbf{T}$ sau $\mathbf{T} \times \mathbf{T} \rightarrow \mathbf{T}$).

Într-un limbaj de programare un *tip de date* reprezintă un *model matematic*. Astfel tipul **int** , caracterizat prin mulţimea finită de întregi **(-32768, 32767)** şi operatorii binari **+, -, *, /** şi **%** ilustrează conceptul de *număr întreg*.

Modelele matematice care nu au reprezentare directă prin tipuri predefinite se pot reprezenta prin *tipuri definite de utilizator* (numite şi *tipuri de date abstracte - TDA*).

2. Declararea claselor

O *clasă* reprezintă un *tip definit de utilizator*. Declararea unei clase se face într-o manieră asemănătoare declarării structurilor şi conţine atât *date* cât şi *funcţii* (metode) şi putem declara variabile de acest tip nou. Un *obiect* este un *exemplar* sau o *instanţă* a unei clase (în vechea terminologie obiectul este echivalent unei variabile, iar clasa este echivalentul unui tip definit de utilizator).

Termenul de ***OOP*** provine din limba engleză şi are următoarea semnificaţie: ***Object Oriented Programming – Programare orientată spre obiect***. Acest domeniu al programării orientate spre obiect are la bază unificarea a două concepte şi anume *datele aplicaţiei* şi *codul necesar prelucrării lor*. Pentru a ajunge la acest scop se oferă utilizatorului câteva facilităţi care permit definirea unor tipuri de date proprii şi a unor operatori care să utilizeze aceste tipuri de date. Astfel se vor obţine tipuri de date care se comportă la fel ca şi tipurile de date şi operatorii standard (tipurile de date: întregi, reale, caracter şi respectiv, operatorii: adunare, scădere, înmulţire, împărţire, ş.a.m.d.). De exemplu, un programator poate creea tipurile de date matrice şi complex, pentru a putea scrie programe în care să poată prelucra numere complexe şi să poată efectua operaţii cu matrici. Astfel pot apărea următoarele declarări şi respectiv operaţii:

```
complex a, b, c;
matrice p, q, r;
. . .
c = a + b;
p = q * r;
```

Tipurile de date “complex” şi “matrice” se numesc **CLASE**; variabilele a, b, c, p, q, r se numesc **OBIECTE** (sau *instanţieri ale claselor* “complex” şi “matrice”), iar operatorii “+” şi “*” se numesc **METODE** ale aceloraşi clase. Prin unificarea datelor înţelegem că o **CLASA** conţine atât structurile de date necesare descrierii unui **OBIECT**, cât şi **METODELE(funcţiile)** cu ajutorul cărora obiectele în cauză vor fi manipulate. În acest mod creşte gradul de abstractizare dar programele devin mai uşor de înţeles, de depanat şi de întreţinut.

Din punct de vedere istoric, primele limbaje care au folosit noţiunea de **CLASA**, au fost SIMULA (1965), SIMULA-2 (1967) şi SMALLTALK (1970). Ele nu s-au putut impune pe piaţa limbajelor de programare deoarece erau limbaje de sine stătătoare şi mulţi dintre programatori nu erau dornici să înveţe un nou limbaj, în defavoarea limbajelor din acel moment. Astfel de abia în anul 1980, când deja limbajul C era acceptat, un colectiv de programatori condus de Bjarne Stroustrup aduce conceptul de **CLASA** într-un dialect al limbajului C, numit *C with Classes*, iar in 1983 ia naştere un nou limbaj numit ***C*++.** Pornind de la multitudinea domeniilor de aplicaţie – de la grafică interactivă, la proiectarea interfeţelor utilizator şi de la exploatarea reţelelor de calculatoare, la tehnicile de proiectare a compilatoarelor – limbajul C++ are un impact foarte mare şi se impune pe piaţa limbajelor de programare obiectuale.

2.1. Tipuri de metode

Metodele pot avea acces la starea obiectului şi de aceea se pot comportă diferit, în funcţie de acţiunea acestora asupra stării obiectului. Astfel avem:

1. Constructori – metode care creează noi instanţieri ale clasei (adică noi obiecte). Spunem că *s-a nascut* un nou obiect atunci când este folosit

constructorul unei clase. Este posibil să creem obiecte în mai multe moduri, deoarece unei clase îi pot corespunde mai mulţi constructori.

2. Destructori – metode care elimină obiecte din memorie.

3. Selectori – metode care nu modifică starea obiectului, asemănătoare cu accesul read-only. Selectori pot invoca obiecte din mai multe clase.

4. Modificatori – metode care modifică starea obiectului, efectuând operaţii de scriere la nivelul datelor obiectului.

2.2. Clase

Clasele sunt de fapt implementarea unor tipuri abstracte de date (***Abstract Data Type - ADT***). O caracteristică importantă a unui limbaj de programare este aceea de a oferi programatorului posibilitatea de a-şi construi tipuri de date ca cele standard, acestea numindu-se tipuri definite de utilizator (*user defined*). În limbajul C se poate folosi cuvântul cheie ***typedef*** prin intermediul căruia se poate defini orice tip de date utilizator. Se mai poate folosi împreună cu tipul structură (***struct***) obţinându-se astfel noi tipuri de date. Cu toate acestea, operaţiile care se pot utiliza asupra acestor noi tipuri de date rămâne tot cel standard, acestea acţionând numai asupra tipurilor standard. Acest aspect se poate rezolva, în POO, prin supraîncărcarea operatorilor (***overloading operators***).

Introducerea tipurilor abstracte de date oferă o anumită stuctură şi unitate prin punerea laolaltă a celor două elemente, şi anume, datele şi operaţiile.

O clasă reuneşte datele şi operaţiile în cadrul aceluiaşi tip de data. În plus, există posibilitatea protejării elementelor componente, atât date cât şi funcţii membre, protejarea nu împotriva necunoaşterii lor de către utilizator ci mai ales împotriva distrugerii lor accidentale. Acestă proprietate se numeşte **încapsulare**.

Deci o **clasă = date + operaţii**.

Într-un program bazat pe programarea standard – programarea modulară, programul foloseşte datele şi operaţiile, în nici un caz un tip de date.

C++ se distinge de limbajele POO pure prin faptul că permite controlul accesului atât la datele membre (încapsulare selectivă a datelor), cât şi la funcţiile membre ale unei clase.

În acest scop, se pot utiliza specificatorii de control: **public**, **private** şi **protected**. Pentru domeniul în care este valabilă declaraţia obiectului, efectul specificatorilor asupra accesului la un membru este:

1. public – membrul poate fi accesat de orice funcţie din domeniul declaraţiei clasei

2. private – membrul este accesibil numai funcţiilor membre şi 'prietene' ale clasei (funcţiile prietene ale unei clase vor fi prezentate ulterior)

3. protected – similar cu **private**, dar accesul se extinde pentru functiile membre şi prietene ale claselor derivate din clasa respectivă (clasele derivate vor fi tratate ulterior)

O clasă poate fi declarată, asemănător cu tipul **struct** sau **union**, folosind cuvântul cheie **class**, astfel:

```
class <nume_clasa> <:lista_clase>
{
  <lista_membri>
}<lista_variabile> ;
```

Unde:

- nume_clasă este numele tipului clasa ales de către utilizator şi care trebuie să fie unic

- lista_clase este lista claselor din care este derivată clasa respectivă (dacă este cazul)

- lista_membri reprezintă secvenţa de declaraţii ale membrilor clasei. Lista conţine declaraţii de date membre şi declaraţii de funcţii membre.

Datele membre pot fi de orice tip, mai puţin tipul clasa derivat (se admit însă pointeri la tipul clasa derivat).

- lista_variabile este lista numelor de variabile alese pentru clasa respectivă

Deşi nume_clasa şi lista_variabile apar ca opţionale, ca şi în cazul tipurilor din limbajul C, **struct** şi **union**, cel puţin una din specificaţii trebuie să existe. De regulă nume_clasa nu se omite, pentru a putea declara ulterior obiecte de acest tip. La declararea obiectelor este suficient să se specifice numele clasei fără cuvântul cheie **class**.

Exemplu:

// definirea clasei matrice

class matrice

{

<lista_membrii>

};

//declararea variabilelor de tip matrice

matrice a,b,c;

Observaţie:

Membrii unei clase au implicit atributul de acces **private**. Metodele asociate datelor trebuie să fie însă accesibile utilizatorului clasei, astfel încât trebuie să fie declarate **public**. Se admite că în definiţia clasei să apară doar prototipurile funcţiilor membre. Definiţiile efective ale funcţiilor se pot face ulterior, în orice alt loc în cadrul programului.

Prezentăm în continuare un exemplu de Tip Abstract de Date(TAD), implementat în limbajul C:

Se observă că în acest program, datele şi funcţiile nu comunică între ele, chiar dacă au fost create pentru un acelaşi obiect – persoana p.

```
#include <stdio.h>
#include <conio.h>


struct persoana
{
char nume[40];
int varsta;
long int telefon;
};
void afiseaza_pers(persoana *p)
{
clrscr();
printf("Numele: %s\n",p->nume);
printf("Varsta: %3d\t Telefonul:%ld\n",p->varsta,p->telefon);
getch();
}
void citeste_pers(persoana *p)
{
clrscr();
printf("Dati numele: ");scanf("%s",p->nume);
fflush(stdin);
printf("Dati varsta: ");scanf("%d",&p->varsta);
fflush(stdin);
printf("Dati numarul de telefon: ");scanf("%ld",&p->telefon);
```

```
fflush(stdin);
}
void main(void)
{
persoana p;
citeste_pers(&p);
afiseaza_pers(&p);
}
```

Folosind clasele, putem aborda aceeaşi problemă, prin de definirea unui nou tip de dată, care va reuni atât datele (care sunt **variabile de instantiere**), cât şi funcţiile (care sunt **funcţii membre**).

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
class persoana{
public:
char nume[40];
int varsta;
long int telefon;
void afiseaza();
void citeste();
};
void persoana::afiseaza()
{
int i;
for(i=0;i<80;i++) printf("#");
printf("\n Numele: %40s",nume);
```

```
printf("\n Varsta: %d",varsta);
printf("\n Telefon: %ld",telefon);
}
void persoana::citeste()
{
clrscr();
printf("Numele: ");scanf("%s",nume);fflush(stdin);
printf("Varsta: ");scanf("%d",&varsta);fflush(stdin);
printf("Telefon:");scanf("%ld",&telefon);fflush(stdin);
}
void main(void)
{
persoana p;
p.citeste();
p.afiseaza();
}
```

Observaţie:

Definiţia unei clase constă din două părţi: **declaraţia** şi **implementarea sa**.

Declaraţia este o înşiruire a elementelor componente ale unei clase (date şi funcţii)

Implementarea constă în acea secţiune în care se implementează funcţiile declarate ca aparţinând clasei. Fiind membre ale unui tip de date, accesul la aceste elemente se realizează prin intermediul operatorului de apartenenţă **‘.’**.

În cadrul definiţiei, cuvântul cheie **public** indică gradul de libertate la acces la elementele componente ale clasei. Prin **public**, toate elementele sunt disponibile în exteriorul clasei, o astfel de clasă fiind asemănătoare cu tipul de date **struct**, însă asemănările se opresc aici.

În schimb utilizând cuvintele cheie **private** sau **protected**, elementele cărora li s-a ataşat aceste cuvinte cheie pot fi accesate numai prin intermediul funcţiilor membre sau *'prietene'*.

Prezentăm în continuare un exemplu de clasă care implementează, o structură de tip listă, în mod special - o structură de tip coadă - **FIFO (First In First Out)**. Obiectele acestei clase sunt structuri de date controlate cu un set de funcţii care asigură următoarele operaţii specifice:

- iniţializarea listei

- adăugarea(introducerea) unui element

- ştergerea(extragerea) unui element

- verificarea dacă lista este vidă

- verificarea dacă lista este plină

Astfel, structura de date, împreună cu metodele de operare asociate reprezintă un ansamblu cu o funcţionalitate bine definită.

Clasa conţine câmpurile:

- *tab[100]*=un vector definit static care conţine elementele listei

- *nrcrt*=numărul curent(actual) de elemente aflate la un moment dat în lista

- *prim*=un număr care reprezintă poziţia primului element din lista

- funcţia *init()*=introduce o valoare de pornire pentru fiecare din cele două variabile specifice, nrcrt respectiv prim

- funcţia *adaug()*=adaugă un element în listă

- funcţia *extrag()*=extrage un element din listă

- funcţia *lista_vida()*=returnează 0 dacă lista este vidă, sau 1 în caz contrar

- funcţia *lista_plina()*=returnează 0 daca lista este plină, sau 1 în caz contrar

Codul sursă al programului este următorul:

// fişierul ”coada.cpp”

```
#include<iostream.h>
#include<conio.h>
//lista de tip coada = FIFO (First In First Out) cu 100 de elemente
class fifo{
// declaratii de tip privat
int tab[100]; // vectorul de 100 de elemente
int nrcrt; // numarul curent de elemente din lista
int prim; // primul element din lista
public:
// declaratii de tip public
int adaug(int); //adauga un element la sfarsit
int extrag(int&);//extrage un element de la inceput
// functii inline
void init() //initializez lista
{
nrcrt=prim=0;
}
int lista_vida() // verific daca stiva este sau nu vida
```

```
{
return(nrcrt>0);//daca nrcrt=0 atunci lista este vida
}
int lista_plina()
{
return(nrcrt<100);//daca nrcrt=100 atunci lista este plina
}
};
int fifo::adaug(int k) //definitia functiei membru este in afara declaratiei
{ //clasei resp. si deci trebuie specificat ca apartine
// clasei fifo prin folosirea operatorului de
// rezolutie ::
if(lista_plina())
{
tab[(prim+nrcrt)%100]=k;
nrcrt++;
cout<<"Lista are "<<nrcrt<<" elemente"<<endl;
return 1;
}
else
{
cout<<"Lista este plina"<<endl;
return 0;
}
}
int fifo::extrag(int &k)
{
```

```
    if(lista_vida())
    {
    k=tab[prim];
    prim=(prim+1)%100;
    nrcrt--;
    cout<<"Lista mai are "<<nrcrt<<" elemente"<<endl;
    return 1;
    }
    else
    {
    cout<<"Lista este vida"<<endl;
    return 0;
    }
    }
    void main(void)
    {
    int k,i;
    fifo lista; //declaratia unui obiect de tip fifo avand numele lista
    fifo *plista;// declaratia unui pointer la lista
    lista.init();//initializeaza lista respectiva
    plista=&lista;//pointerul primeste adresa obiectului lista
    cout<<"Dati un element care se va adauga in lista ";
    cin>>k;
    plista->adaug(k);// adauga elementul cu valoarea k in lista;
    //afisarea elementelor din lista: for(i=prim;i<nrcrt;i++) cout<<tab[i]
<<" ";
    getch();
    }
```

Definiţi şi implementaţi clasa **Cerc**, având ca dată membră **Raza**, un constructor şi ca funcţii membre **Aria** şi **Circumferinta**.

```
#include<iostream.h>
#include<conio.h>
#define PI 3.14
class cerc{
float raza;
public:
cerc();
float aria();
float circumferinta();
};
cerc::cerc()
{
cout<<"Raza cercului este = ";
cin>>raza;
}
float cerc::aria()
{
```

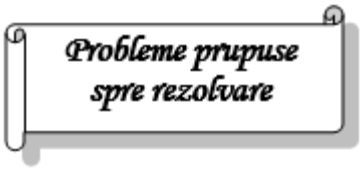
```
return (PI*raza*raza);
}
float cerc::circumferinta()
{
return (2*PI*raza);
}
void main(void)
{
cerc c;
clrscr();
cout<<"Aria cercului este = "<<c.aria()<<endl;
cout<<"Circumferinta cercului este = "<<c.circumferinta()<<endl;
getch();
}
```

Definiţi şi implementaţi clasa **Cilindru**, având ca date membre **Raza** şi **Inaltimea** cilindrului şi ca funcţii membre: un constructor, **Aria** şi **Volumul**.

```
#include<iostream.h>
#include<conio.h>
#define PI 3.14
class cilindru{
float raza;
```

```
float inaltimea;
public:
cilindru();
float aria();
float aria_totala();
float volum();
};
cilindru::cilindru()
{
cout<<"Dati raza cilindrului = ";cin>>raza;
cout<<"Dati inlatimea cilindrului = ";cin>>inaltimea;
}
float cilindru::aria()
{
return (2*PI*raza*inaltimea);
}
float cilindru::aria_totala()
{
float a;
a=aria();
return(a+2*PI*raza*raza);
}
float cilindru::volum()
{
return (PI*raza*raza*inaltimea);
}
void main(void)
{
```

```
clrscr();
cilindru c;
cout<<"Aria cilindrului este = "<<c.aria()<<endl;
cout<<"Aria totala este = "<<c.aria_totala()<<endl;
cout<<"Volumul cilindrului este = "<<c.volum()<<endl;
getch();
}
```



1. Definiţi şi implementaţi clasa **Televizor**, având ca date membre: **Volumul** şi **Canalul** şi ca funcţii membre: un constructor, **Pornire**, **Oprire**, **Schimbare_Volum**, **Schimbare_Canal**.

2. Definiţi şi implementaţi clasa **Dreptunghi**, având ca date membre: **Lungimea** şi **Laţimea** şi ca funcţii membre: un constructor, **SetLungime**, **SetLaţime**, **GetLungime**, **GetLaţime**, **Arie** şi **Perimetru**.

Capitolul 2 - Constructori şi destructori

2.1. Operatorul de rezoluţie

Întrucât definirea funcţiilor se va face în afara domeniului de definiţie al clasei, numele funcţiei trebuie să fie însoţit de numele clasei, şi să fie separat de aceasta prin *operatorul de rezoluţie* sau *vizibilitate* (**::**). Aşadar, implementarea clasei, (fişierul **"punct.cpp"**) va fi:

```
// fişierul "punct.cpp"
#include <iostream.h>
#include <iomanip.h>
void punct::init(double x, double y){
x0 = x;
y0 = y;
};
void punct::setx(double x){ x0 = x; };
void punct::sety(double y){ y0 = y; };
double punct::getx(){ return x0; };
double punct::gety(){ return y0; };
void punct::afisare(){
long f = ios::fixed;
cout << setiosflags(f) << setw(6) << setprecision(2);
cout << "(" << x0 << "," << y0 << ")" << endl;
};
};
```

De obicei declararea clasei (interfaţa) şi definirea ei (implementarea) se fac în fişiere separate. Declaraţia unei clase se face într-un *fişier antet*. Un fişier antet trebuie inclus o singură dată în fişierul programului.

Pentru a evita includerea multiplă de fişiere antet se utilizează directive **#ifndef** sub forma:

#ifndef _simbol_

#define _simbol_

declaraţie clasă (fişier antet)

#endif

Programul care utilizează clasa **punct** este un program constituit din mai multe fişiere, care vor alcătui un *proiect*. El va conţine o directivă de includere a fişierului de interfaţă **punct.h** pentru definirea clasei. La prima întâlnire a fişierului antet, simbolul nu este definit, incluzându-se fişierul antet. După prima includere, simbolul fiind definit se ignoră porţiunea cuprinsă între **#define** şi **#endif**.

Fişierul de implementare conţine definiţiile metodelor, directiva de includere a fişierului antet al interfeţei şi alte directive de includere de fişiere.

#include <iostream.h>

#include "clasa.h"

// definiţii funcţii membre ale clasei

Programul utilizator (al treilea fişier) va conţine: includerea diverselor fişiere antet, includerea fişierului antet al clasei şi funcţia **main()**.

#include <iostream.h>

#include "clasa.h"

```
void main(){
. . .}
```

2.2. Funcţii membre inline

Funcţiile mici, cu puţine instrucţiuni, apelate frecvent se compilează ineficient (se rezervă spaţiu în stivă pentru parametrii şi rezultat, se efectuează salt la funcţie şi apoi salt pentru revenire). Mult mai eficientă ar fi expandarea apelului funcţiei prin corpul ei, ca în macroinstrucţiuni. Apelul unei funcţii declarate **inline** este înlocuit la compilare prin corpul funcţiei (apel realizat prin expandare). O funcţie inline nu trebuie să conţină bucle.

Funcţiile membre definite în interiorul clasei sunt în mod implicit **inline**, dar este posibil ca şi funcţii definite în afara clasei să fie declarate explicit **inline** (expandate).

```
class Data{
public:
int zi();
. . .
private:
int a, l, z;
};
// funcţie declarată în mod explicit inline în-afara clasei
inline int Data::zi(){return z;};
```

```
// definirea clasei – fişierul punct.h
class punct{
private:
double x0, y0; // date membre
public:
inline void init(double x=0, double y=0); //nu era necesară
inline void setx(double x=0){ x0=x; }; //declararea explicită
inline void sety(double y=0){ y0=y; }; //erau considerate inline
double getx(){ return x0; }; // este implicit inline
double gety(){ return y0; };
void afisare();
};


// fisierul punct.cpp
#include <iostream.h>
#include <iomanip.h>
//deşi este definită în afara clasei este inline (apel expandat)
inline void punct::init(double x, double y){
x0 = x;
y0 = y;
};
//nu este inline (apel=salt cu revenire)
void punct::afisare(){
long f = ios::fixed;
cout << setiosflags(f) << setw(6) << setprecision(2);
```

cout << "(" << x0 << "," << y0 << ")" << endl;

};

};

2.3. Referinţe

O referinţă se comportă ca un pointer constant, care este în mod automat dereferenţiat. Referinţele se folosesc:

- în listele de parametri ale funcţiilor

- ca valori de întoarcere ale funcţiilor

La creerea unei referinţe, aceasta trebuie iniţializată cu adresa unui obiect (nu cu o valoare n constantă).

int x;

int& rx = x; //referinţa iniţializată cu o adresă

Referinţa poate fi privită ca un *pointer inteligent*, a cărui *iniţializare* este *forţată* de către compilator (la definire) şi care este *dereferenţiat automat*.

Spre deosebire de pointeri:

- referinţele sunt iniţializate la creere (pointerii se pot iniţializa oricând)

- referinţa este legată de un singur obiect şi această legătură nu poate fi modificată pentru un alt obiect

- nu există referinţe nule – ele sunt totdeauna legate de locaţii de memorie.

2.4. Constructori şi destructori

Pentru crearea, initializarea, copierea si distrugerea obiectelor, in C++ se folosesc functii membre speciale, numite **constructori** si **destructori**:

Functia de tip ***constructor*** se apeleaza automat la crearea fiecarui obiect al clasei, creare care poate fi statica sau dinamica (folosind operatorul **new**).

Functia de tip ***destructor*** se apeleaza automat la eliminarea unui obiect al clasei, la incheierea timpului de viata sau, in cazul variabilelor dinamice, este solicitat prin program (cu operatorul **delete**).

Aceste tipuri de functii se declara si se definesc asemanator cu celelalte functii membre, dar au cateva particularitati care le diferentiaza de acestea prin:

- Numele functiilor *constructor* sau *destructor* coincide cu numele clasei careia ii apartine;In plus destructorii se diferentiaza prin faptul ca numele lor este precedat de caracterul “~”.

- In declaratiile si definitiile constructorilor si destructorilor nu se specifica nici un tip de rezultat, nici macar tipul void.

- Constructorii si destructorii nu pot fi mosteniti, dar pot fi apelati de clasa derivata.

- Nu se pot utiliza pointeri catre constructori sau destructori.

- Constructorii pot avea parametri, inclusiv parametri impliciti, si pot fi supradefiniti.

- Destructorii nu au aceste proprietati. Un constructor fara parametri se numeste ***constructor implicit***.

Daca o clasa nu dispune de constructori si destructori definiti, compilatorul va genera automat un constructor implicit, respectiv un destructor, care sunt si functii publice.

Acestia au fost utilizati pentru crearea si eliminarea obiectelor fifo in programul din cursul precedent.

De obicei, constructorii si destructorii se declara publici, deoarece, daca s-ar declara privati atunci nu se mai pot declara obiecte de tipul respectiv.

În absenţa unor declarări explicite, compilatorul furnizează următoarele funcţii membre:

- un constructor implicit
- un constructor de copiere
- un operator de atribuire
- un destructor implicit
- un operator de adresare

2.4.1. Crearea, initializarea si eliminarea obiectelor

Unele din exemplele pe care le vom implementa in continuare se vor baza pe clasa fifo declarata anterior.

Constructorul este simplu si poate fi definit **inline**:

```
class fifo{
// declaratii de tip privat
int tab[100]; // vectorul de 100 de elemente
int nrcrt; // numarul curent de elemente din lista
int prim; // primul element din lista
public:
// declaratii de tip public
// functii inline
fifo()
{
nrcrt=prim=0 ;
cout<<”Constructor fifo”<<endl;
}
~fifo()
```

```
{
cout<<"Destructor fifo"<<endl;
}
void init() //initializez lista
{
nrcrt=prim=0;
}
int lista_vida() // verific daca stiva este sau nu vida
{
return(nrcrt>0);//daca nrcrt=0 atunci lista este vida
}
int lista_plina()
{
return(nrcrt<100);//daca nrcrt=100 atunci lista este plina
}
int adaug(int); //adauga un element la sfarsit
int extrag(int&);//extrage un element de la inceput
};
```

Pentru vizualizarea apelarii constructorului, s-a adaugat un mesaj. Clasa nu are nevoie de un destructor, drept pentr care s-a adaugat unul formal, care nu face nimic dar afiseaza un mesaj, de identificare.

Se poate executa urmatorul program care sa contina clasa fifo si care sa aiba doar declaratia unei variabile si functia principala, astfel:

```
fifo lista;
void main()
{}
```

Oricat ar parea de surprinzator, conform celor spuse mai inainte, executia programului va avea ca efect, urmatoarele afisari:

```
Constructor fifo
Destructor fifo
```

Chiar daca functia main() este vida, conform definitiei unei clase, se obtine un astfel de rezultat.

In exemplul urmator se reia declaratia clasei fifo, pentru a obtine liste de dimensiune variabila, care implicit ar fi egala cu 100 de elemente. In acest scop, este necesar ca tabloul sa fie alocat dinamic.

Se fac urmatoarele modificari :

se elimina functia init() si se prevede in schimb un constructor care preia sarcina alocarii dinamice a tabloului cu dimensiunea specificata si pe cea a initializarii corespunzatoare a listei.

constructorul va fi supradefinit pentru a permite utilizarea unor liste cu un numar implicit de 100 de elemente in tablou.

de asemenea este definit un destructor pentru eliberarea spatiului alocat pentru tablou.

Programul va ilustra decalarea si functionarea constructorilor si destructorilor si va pune in evidenta momentele in care acestia se apeleaza, prin afisarea unor mesaje la ecran:

```
#include<iostream.h>
#include<conio.h>
//lista FIFO cu nmax numere de elemente
class fifo{
// declaratii de tip privat
int nmax; // dimensiunea listei
int *tab; // adresa tabloului listei
int nrcrt; // numarul curent de elemente din lista
```

```
int prim; // primul element din lista
public:
// declaratii de tip public
fifo(int); // constructor 1
fifo(); // constructor 2
~fifo(); // destructor
int adaug(int); //adauga un element la sfarsit
int extrag(int&);//extrage un element de la inceput
// functii inline
int lista_vida() // verific daca stiva este sau nu vida
{
return(nrcrt>0);//daca nrcrt=0 atunci lista este vida
}
int lista_plina()
{
return(nrcrt<nmax);//daca nrcrt=nmax atunci lista este plina
}
};
// definitiile functiilor membre ale clasei fifo
// constructor 1, lista de n elemente
fifo::fifo(int n)
{
nmax=n;
tab=new int;
nrcrt=prim=0;
cout<<"Constructor 1,lista de "<<nmax<<"elemente "<<endl;
}
//constructor 2, implicit
```

```
fifo::fifo()
{
//lista de 100 elemente
nmax=100;
tab=new int;
nrcrt=prim=0;
cout<<"Constructor 2, lista de 100 de elemente"<<endl;
}
//destructor
fifo::~fifo()
{
cout<<"Destructor, lista de "<<nmax<<" elemente"<<endl;
delete tab;
}
int fifo::adaug(int k) //definitia functiei membru este in afara declaratiei
{ //clasei resp. si deci trebuie specificat ca apartine
// clasei fifo prin folosirea operatorului de
// rezolutie ::
if(lista_plina())
{
tab[(prim+nrcrt)%nmax]=k;
nrcrt++;
cout<<"Lista are "<<nrcrt<<" elemente"<<endl;
return 1;
}
else
{
```

```
cout<<"Lista este plina"<<endl;
return 0;
}
}


int fifo::extrag(int &k)
{
if(lista_vida())
{
k=tab[prim];
prim=(prim+1)%nmax;
nrcrt--;
cout<<"Lista mai are "<<nrcrt<<" elemente"<<endl;
return 1;
}
else
{
cout<<"Lista este vida"<<endl;
return 0;
}
}
void main(void)
{
int k,i;
fifo *plista;// declaratia unui pointer la lista
plista=new fifo(20); //creare obiect dinamic
delete plista;
```

```
getch();
}
```

2.4.2. Redefinirea functiilor member

Exemplul urmator prezinta o clasa **sir** care defineste functii membre de comparare a doua siruri, dintre care una din ele va fi rescrisa.

In acest exemplu, singura data de tip privat este pointerul la un sir de caractere *text. Constructorul **sir** si functia *compar()* sunt numai declarate in cadrul clasei, pe cand destructorul **~sir** si functia denumita **afiseaza()** sunt definite in mod **inline**. Functia **compare()** este redefinita. Ea are doua sabloane (prototipuri). Primul sablon este utilizat cand se compara doua siruri de lungimi diferite sau nu, iar al doilea cand se compara numai primele *numarcar* caractere ale acestora. Notatiile **sir &sir1** si **sir &sir2** reprezinta referintelor celor doua siruri ce trebuie comparate.

Functia **compar()** intoarce un rezultat de tip intreg, astfel incat prima definitie a functiei compara doua siruri de lungime diferita, iar cea de-a doua compara primele numarcar caractere din cele doua siruri.

Valoarea compararii cu functiile specializate **strcmp()** si **strncmp()** se obtine prin scaderea nedistructiva a caracterelor celor doua functii aflate pe acceasi pozitie relativa. Se pot obtine urmatoarele valori :

Daca s1 < s2 atunci rezultatul este <0

Daca s1 = s2 atunci rezultatul este =0

Daca s1 > s2 atunci rezultatul este >0.

```
#include<iostream.h>
#include<conio.h>
```

```
#include<string.h>
#include<stdio.h>
class sir{
// declaratie de tip privat
char *text;
public:
// declaratii de tip public
sir(char *sirul); // constructor
~sir(){delete text;} // destructor
int compar(sir &s1, sir &s2);
// functia membra compar va fi redefinita
int compar(sir &s1, sir &s2, unsigned int numarcar);
void afiseaza(char *mesaj=" ")
{
cout<<mesaj<<text<<endl;
}
};
sir::sir(char *sirul) // definirea constructorului
{
text= new char[strlen(sirul)];
strcpy(text,sirul);
}
int sir::compar(sir &s1,sir &s2)
{
return strcmp(s1.text,s2.text);
}
int sir::compar(sir &s1,sir &s2, unsigned int numarcar)
{
```

```
return strncmp(s1.text,s2.text,numarcar);
}
void main(void)
{
sir sir1("abcd"), sir2("abcdef");
sir1.afiseaza("Primul sir este: ");
sir2.afiseaza("Al doilea sir este: ");
int rezultat1,rezultat2;
rezultat1=sir1.compar(sir1,sir2);
cout<<"Rezultatul compararii este: "<<rezultat1<<endl;
rezultat2=sir1.compar(sir1,sir2,4);
cout<<"Rezultatul compararii este: "<<rezultat2<<endl;
}
```

2.5. Pointerul this

O funcţie membru al unei clase, apelată dintr-un obiect, pentru a accesa membrii date ai obiectului, va avea un parametru implicit pointer la obiect. Acest pointer poartă numele de pointerul **this**, şi se declară într-o clasă **C** ca: **C* const this;**

Fiind cuvânt cheie, el nu poate fi declarat explicit; fiind un pointer constant el nu poate fi modificat. Referirea la datele membre x0 şi y0 se face prin intermediul acestui pointer ascuns ca **this**->x0 şi **this**->y0. O funcţie membru care returnează un obiect o poate face prin return ***this** (returnarea unui pointer la obiect se face prin return this).

Capitolul 3 - Tablouri de obiecte

3.1. Tablouri de obiecte

La fel cum exista posibilitatea de a crea tablouri de structuri, tot astfel exista posibilitatea de a crea si tablouri de obiecte. De exemplu, putem avea un triunghi vazut ca un vector de trei puncte :

Punct triunghi[3] ;

Putem de asemenea, initializa acest vector chiar pe linia de declarare:

Punct triunghi[3]={{0,0},{20,30},{40,70}};

In plus, putem initializa un tablou de obiecte chiar in cadrul declararii sale:

Punct triunghi[3]= { Punct (1,2),

Punct(30,90),

Punct (40,40)

};

unde clasa Punct are urmatoarea declaratie:

```
class Punct {
      public:
            int x,y;
            Punct(int x1, int y1)  {    x=x1 ;     y=y1;}
            Punct(int x1)       { x=x1 ;     y=x1;}
};
```

3.2. Membri statici ai unei clase

Spre deosebire de **‘auto’**, **‘register’** si **‘extern’**, cuvantul cheie **‘static’** poate fi utilizat in prefixarea membrilor unei clase. Evident ca, odata declarat **‘static’**, membrul respectiv va avea proprietati diferite de membrii obisnuiti (ne-statici). Aceste proprietati apar datorita faptului ca *membrii statici nu apartin unui obiect anume, si sunt comuni tuturor instantierilor(obiectelor) unei clase*.

Sunt situatii de programare cand se doreste ca un singur spatiu de memorie sa fie utilizat de toate obiecte unei clase. In limbajul C, am putea utiliza o variabila globala, dar acest lucru nu este chiar sigur. Variabilele globale pot fi modificate de catre oricine, iar numele lor se pot potrivi(suprapune) cu alte nume identice in cazul unui proiect mai mare. Ar fi ideal ca o variabila(data, informatie) sa poata sa fie stocata (memorata) ca si cum ar fi globala, dar sa fie ascunsa intr-o clasa, si asociata clar cu o clasa.

Acest aspect se poate rezolva folosind date membri intr-o variabila de tip static in interiorul unei clase.

Declararea unei date statice se face in interiorul unei clase, iar definitia trebuie sa se faca in afara clasei (nu se accepta functii inline in acest caz) si este permisa doar o singura definitie.

Sintaxa declarari unei date statice este obisnuita:

De exemplu:

```
class A {
static int i;
public:
// . . . .
};
```

iar mai tarziu, definitia: int A::i=1;

Este asemanatoare cu declararea unei variabile globale: int i=1;

doar ca in cazul claselor se foloseste operatorul de rezolutie specific A::i.

Se poate induce ideea ca A::i este **private** si totusi poate fi manipulat in afara, de unde s-ar putea trage concluzia ca mecanismul protectiei ar fi incalcat. Totusi nu este asa, din doua motive:

- in primul rand, este singurul loc unde aceasta initializare este acceptata de compilator. Intr-adevar daca o variabila statica ar avea un constructor atasat, am putea chema constructorul pentru crearea sa in loc sa utilizam operatorul '='.

- in al doilea rand, odata ce a fost facuta definitia, utilizatorul nu mai poate crea o a doua definitie, deoarece link-editorul va raporta o eroare. Si atunci creatorul clasei este fortat sa creeze definitia, altfel nu se va face legatura in timpul executiei programului. Aceasta ne asigura ca aceasta definitie se poate face doar de catre creatorul clasei respective.

```
#include<iostream.h>
int x=100;
class cu_static{
static int x;
static int y;
public:
void afiseaza()
{
```

```
cout<<"cu_static::x = "<<x<<endl;
cout<<"cu_static::y = "<<y<<endl;
}


};
int cu_static::x=1;
int cu_static::y=x+1;
int main()
{
cu_static a;
a.afiseaza();
cout<<"variabila globala x = "<<x<<endl;
}
```

Dupa executia programului se va afisa :

```
cu_static::x = 1
cu_static::y = 2
variabila globala x = 100
```

3.3. Manevrarea dinamica a obiectelor

Obiectele, ca si celelalte tipuri de date ale limbajului C++, pot fi alocate dinamic, adica se poate aloca memorie doar in timpul executiei programului.

In limbajul C, pentru aceste actiuni puteam folosi functii predefinite, din fisierul ***alloc.h*** – *calloc()*, *malloc()*, *realloc()* si *free()*. Dintre acestea cele mai utilizate sunt *malloc()* si *free()*. In limbajul C++, echivalentul celor doua functii specificate mai sus sunt cei doi operatori ***new*** si ***delete***.

3.3.1 Operatorul *new*

Avantajul acestui operator, fata de functia malloc() consta in doua aspecte:

- *new* stie cata memorie trebuie alocata oricarui tip de variabila, spre deosebire de functia *malloc()*, unde trebuia sa specifice utilizatorul aceasta dimensiune;

- *new* poate produce apelul unui constructor, in timp ce malloc() nu are aceasta proprietate.

Exemple:

```
// varianta C
int *pointer=(int*)malloc(sizeof(int));
// varianta C++
int *pointer= new int;
```

Operatorul ***new*** se poate utiliza si pentru declararea de tablouri:

int *p=new int[10];

Folosind acest operator la clase se pot face urmatoarele tipuri de declarari:

Punct *p=new Punct(20,20);

Efectul este rezervarea de memorie pentru obiectul specificat (Punct), dupa care de va apela constructorul obiectului.

Asemanator se poate declara un tablou de obiecte:

Punct *poligoane=new Punct[10];

3.3.2 Operatorul *delete*

Operatorul **delete** este analogul functiei **free()** din limbajul C, si are ca efect eliberarea spatiului de memorie ocupat cu ajutorul operatorului ***new***.

Asa cum operatorul ***new*** este utilizat de obicei in interiorul constructorilor, tot asa si operatorul ***delete*** va fi utilizat in interiorul destructorilor.

Cu toate ca ***delete*** permite eliberarea selective a spatiului alocat, acest operator nu este la fel de puternic la operatorul ***new***.

Din urmatorul exemplu, am putea trage concluzia falsa ca folosirea operatorului ***delete*** va conduce la eliberarea spatiului de memorie alocat pentru intreg tabloul de numere:

int *m=new int[10];

delete m;

De fapt, s-a eliberat spatiul de memorie doar pentru in singur numar intreg – m[0], restul ramanand ocupat. Pentru eliberarea completa a intregului spatiu alocat cu operatorul ***new***, trebuie sa folosim operatorul ***delete*** astfel:

int *m=new int[10];

delete[] m;

3.4. Pointeri catre membrii unei clase

Un *pointer* este o variabila care retine adresa unei anumite zone de memorie, in care poate fi ori o data(informatie), ori o functie. In limbajul C++, un pointer catre un membru al unei clase urmeaza acelasi principiu, cu exceptia faptului ca selecteaza o locatie din interiorul unei clase. Problema ar putea fi faptul ca toti pointerii au o adresa, dar in interiorul unei clase nu exista o adresa, de unde selectarea unui membru sa fie posibila.

Astfel pointerul respectiv nu va contine adresa de memorie a intregii clase, ci numai deplasarea datei in cadrul oricarui obiect al clasei.

Pentru operatii cu pointeri exista cateva reguli de sintaxa:

- pentru preluarea adresei unui membru se poate folosi operatorul '**&**' urmat de numele clasei, operatorul de rezolutie si numele membrului:

pointer_membru=&clasa::membru;

- pentru accesul la un membru dupa preluarea adresei, trebuie indicat un obiect al clasei si numele variabilei pointer, folosind unul dintre operatorii specializati:

1. operatorul **.*** daca se specifica un obiect al clasei
2. operatorul **->*** daca se specifica un pointer de obiect

Sintaxa este urmatoarea:

obiect.*pointer

sau

pointer_obiect->*pointer_membru

Pentru exemplificare prezentam o clasa numita pozitie care se foloseste pentru a stabili pozitia unui punct in plan :

```
#include<iostream.h>
class pozitie{
public:
int x,y; //coordonatele unui punct
// functii membre
pozitie(int abcisa=0, int ordonata=0) // constructor
{
x=abcisa;
y=ordonata;
cout<<"Constructor ";
afisare();
}
void deplasare(int abcisa, int ordonata)
{
x+=abcisa;
y+=ordonata;
}
void afisare() // afisarea coordonatelor
{
cout<<" x = "<<x;
cout<<" y = "<<y<<endl;
}
```

```
};
void main(void)
{
pozitie p(1,1), *pointer;
int pozitie::x_pointer, pozitie::y_pointer;
void (pozitie::*pointer_functie)(int,int);
pointer=&p;
x_pointer=&pozitie::x;
y_pointer=&pozitie::y;
pointer_functie=pozitie::deplasare;
(p.pointer_functie)(10,10);
cout<<"Noua pozitie este : ";
cout<<"("<<p.x_pointer<<","<<p.y_pointer<<")"<<endl;


(p.pointer_functie)(10,10);
cout<<"Noua pozitie este : ";
cout<<"("<<p.x_pointer<<","<<p.y_pointer<<")"<<endl;
}
```

Tablouri de dimensiune variabila

Urmatoarea problema rezolvata prezinta o modalitate de creare a unor tablouri de dimensiuni variabile. Tipul de date il numim **array** (vector=sir) si contine :

un constructor – care creeaza si initializeaza tablou-rile

un destructor – elibereaza memoria alocata pentru un tablou

o procedura de concatenare a doua tablouri

```
#include<iostream.h>
#include<process.h>
#include<conio.h>

class array{
int *pointer; // pointer la tablou=array
int dimensiune; // dimensiunea tabloului
void print(void); // afiseaza tabloulul
void test(void *); // testeaza daca s-a alocat corect memoria
public:
array(int); //constructor
~array(void) //destructor
{
delete[dimensiune] pointer;
}
void concatenare(array &); //procedura de concatenare
int cauta(array &); // cauta un element
};
// procedura de test
```

```
// Testeaza valoarea unui pointer pentru care s-a alocat dinamic memorie.
// Daca pointer-ul este setat la NULL inseamna ca opeartia de alocare dinamica
// a memoriei nu s-a efectuat corect
void array::test(void *p)
{
if(p==NULL)
{
cout<<"Memorie insuficienta"<<endl;
exit(1);
}
}
// se creeaza un tablou de intregi cu "numar_elem" elemente.
// Initializarea tablou-ului cu valori citite de la tastatura.
array::array(int numar_elem)
{
cout<<endl;
cout<<"Introduceti "<<numar_elem<<" numere intregi"<<endl;
// alocarea dinamica a memoriei necesare tablou-ului
dimensiune=numar_elem;
pointer=new int[numar_elem];
// initializare tablou
for(int i=0;i<numar_elem;i++) cin>>pointer[i];
// tiparire tablou
print();
}
// afiseaza elementele dintr-un tablou
```

```
void array::print(void)
{
cout<<endl<<"Sirul este : ";
for(int i=0;i<dimensiune;i++) cout<<pointer[i]<<" ";
cout<<endl;
cout<<"apasati o tasta pentru continuare ";
getch();
}
// concateneaza tablou care a apelat metoda cu cel primit ca parametru
// (prin referinta); rezultatul se va pastra in obiectul care a apelat metoda.
void array::concatenare(array &referinta_array)
{
int *temp;
int dimensiune_finala;
// calcularea noii dimensiuni a tabloului rezultat
dimensiune_finala=dimensiune+referinta_array.dimensiune;
// alocare memorie necesara
temp=new int[dimensiune_finala];


test((void*) temp);
// copierea celor doi "operanzi" in cadrul "rezultatului"
for(int i=0;i<dimensiune_finala;i++)
if(i<dimensiune) temp[i]=pointer[i];
else temp[i]=referinta_array.pointer[i-dimensiune];
// eliberarea memoriei necesare folosite de primul operand
delete[dimensiune] pointer;
```

```
// setare dimensiune corecta
dimensiune=dimensiune_finala;
// setare pointer la tablou
pointer=temp;
// tiparire rezultat
print();
}



void main(void)
{
array a(6);
array b(4);
int i;
cout<<endl;
cout<<"==== Introduceti dimensiunea tabloului: ==== ";
cin>>i;
cout<<endl;
array c(i);
a.concatenare(b);
a.concatenare(c);
}
```

Daca se introduc valorile 1,2,3 pentru tabloul a respectiv valorile 4,5,6,7 pentru tabloul b si valorile 11,22,33,44 pentru tabloul c vom obtine :

a={1,2,3,4,5,6,7} in prima faza,

iar apoi a= {1,2,3,4,5,6,7,11,22,33,44,}.

Capitolul 4 - Supraîncărcarea operatorilor

Operatorii sunt notaţii concise, infixate, pentru operaţii matematice uzuale. Limbajul C++, ca orice limbaj de programare asigură un set de operatori pentru tipurile primitive. În plus, faţă de limbajul C, C++ oferă posibilitatea asocierii operatorilor existenţi cu tipurile definite de utilizator . Astfel, prezintă interes extinderea operatorilor în aritmetică complexă, algebra matricială, în lucrul cu şiruri de caractere, etc. Un operator poate fi privit ca o funcţie, în care termenii sunt argumentele funcţiei (în lipsa operatorului +, expresia **a+b** s-ar calcula apelând funcţia **aduna(a,b)**).

Limbajul C++ introduce următorii operatori noi: **new** şi **delete** - pentru gestiunea memoriei dinamice, operatorul de rezoluţie (**::**) şi operatorii de acces la membri: **.*** şi **->***.

4.1 Supraîncărcare

Prin *supraîncărcarea* unui operator, acesta este utilizat în contexte diferite, avînd aceeaşi semnificaţie sau o semnificaţie diferită. Astfel operatorul + este supraîncărcat pentru adunarea întregilor şi a realilor; operatorul << este supraîncărcat cu semnificaţia de deplasare la stînga, dacă termenii sunt întregi sau inserare în flux, dacă un termen este un flux de date, iar celălalt un obiect.

Programatorul poate asocia noi semnificaţii operatorilor existenţi prin *supraîncărcarea* operatorilor. Vom evita folosirea termenului redefinire, căruia îi vom da o semnificaţie diferită, în contextul moştenirii.

Anumiţi operatori pot fi utilizaţi cu orice tip de termeni (sunt deja supraîncărcaţi global de către compilator). Aceştia sunt: **new, delete, sizeof, ::, =, &, ->*, .*, ., ->.**

Pot fi supraîncărcaţi următorii operatori:

+ - * / % ^ & | ~ ! = < >

+= -= *= /= %= ^= &= |= >>= <<= == != <= >=

&& || ++ -- ->* , -> << >> [] ()

new new[] delete delete[]

Nu pot fi supraîncărcaţi operatorii: **::, ., .*, ?:, sizeof**.

Setul de operatori ai limbajul C++ nu poate fi extins prin asocierea de semnificaţii noi unor caractere, care nu sunt operatori de exemplu nu putem defini operatorul **).

Prin supraîncărcarea unui operator nu i se poate modifica *aritatea* (astfel operatorul **!** este unar şi poate fi redefinit numai ca operator unar. De asemeni *asociativitatea* şi *precedenţa* operatorului se menţin.

La supraîncărcarea unui operator nu se pot specifica *argumente cu valori implicite*.

Operatorii supraîncărcaţi într-o clasă sunt moşteniţi în clasele derivate (excepţie face operatorul de atribuire =).

Semnătura unei funcţii în care se supraîncarcă un operator este:

tip_rezultat operator#(listă_argumente);

4.2 Funcţii prietene

Atunci cand am vorbit despre tipurile abstracte de date, am specificat ca o proprietate de baza este protectia datelor membru ale tipului respectiv. Elementele protejate constituie asa numita implementare a tipului abstract.

Tipurile abstracte de date se definesc cu ajutorul claselor. Se obisnuieste sa se spuna ca datele protejate ale unui tip abstract sunt *incapsulate* in clasa care defineste tipul respectiv.

Protectia datelor se realizeaza prin faptul ca la ele au acces numai functiile membru ale tipului (clasei). De asemenea, daca o fucntie membru este protejata, atunci ea poate fi apelata numai prin intermediul unei functii membru a tipului (clasei).

Acest mod de lucru, desi asigura o protectie buna a elementelor membru protejate ale clasei (elemente protejate prin **private** sau **protected**), uneori este considerat rigid. Astfel, desi exista fucntii descrise in C care pot fi utilizate pentru a prelucra instantieri (obiecte) ale unei clase, ele nu se pot utiliza simplu deoarece nu sunt functii membru ale clasei respective si deci nu au acces la datele membru. Mai mult decat atat o functie membru se apeleaza totdeauna in dependenta cu un obiect care este numit obeictul curent al apelului. In acest fel, o functie membru se apeleaza prin una din urmataoarele formate:

nume_obiect.nume_functie_membru(…)

sau

pointer_nume_clasa->nume_functie_membru(…)

In cazul functiilor obisnuite nu se admit astfel de apeluri, toate datele prelucrate de functie se transfera prin parametri sau sunt globale.

De aceea, o functie obisnuita poate fi utilizata pentru a prelucra obiectele unei clase, daca ea se modifica in asa fel incat sa devina functie membru.

S-a facut un compromis pentru a admite accesul la elementele protejate si pentru anumite functii care nu sunt membru. Aceste functii au fost numite **functii prieten (friend)** pentru clasa respectiva. Ele trebuie precizate ca atare in definitia clasei. In acest scop, prototipurile lor sunt prezente in definitia clasei si sunt precedate de cuvantul cheie **friend**.

Exemplu:

Fie tipul definit de utilizator:

struct complex{

```
double real;
double imag;
};
```

Functia *modul*, pentru calculul modulului unei date de tip complex poate fi definita, ca mai jos, ca o functie obisnuita:

```
double modul(complex *z)
{
return sqrt(z->real * z->real + z->imag * z->imag);
}
```

Daca se considera declaratiile:

```
complex z1={1,1};
double d;
```

atunci instructiunea de atribuire:

```
d=modul(&z1);
```

atribuie lui d modulul numarului complex z1=1+i.

Acelasi lucru se poate realiza implementand tipul abstract complex astfel:

```
class complex{
double real;
double imag;
public:
```

```
complex(double x=0,double y=0)
{
real=x; imag=y;
}
double modul()
{
return sqrt(real->real + imag*imag);
}
….
};
```

Declaratia pentru z1 se scrie:

complex z1(1,1);

iar instructiunea de atribuire devine:

d=z1.modul();

In acest caz, functia *modul()* nu are parametru deoarece ea se apeleaza pentru obiectul z1. In corpul functiei *modul()* este definit implicit pointerul *this* si acesta are ca valoare chiar adresa lui z1.

Expresia:

real*real+imag*imag

din corpul functiei *modul*, trebuie considerata ca si cand ar fi scrisa sub forma:

this->real*this->real + this>imag*this->imag

Functia *modul*, definita pentru tipul utilizator complex (cu struct) poate inlocui functia membru *modul* a clasei complex daca ea se declara ca

functie prieten a clasei complex, astfel:

```
class complex{
double real;
double imag;
public:
complex(double x=0,double y=0)
{
real=x;imag=y;
}
friend double modul(complex *z);
….
};
```

Functia *modul*, prieten al clasei complex, are aceeasi definitie ca si functia *modul* definita pentru tipul complex introdus prin constructia **struct**. De asemenea, ea se apeleaza ca orice functie obisnuita.

In exemplu de fata, daca se considera instantierea:

complex z(1,1);

atunci functia *prieten modul* se apeleaza in mod obisnuit, adica prin:

d=modul(&z);

Spre deosebire de functiile membru, in cazul functiilor prieten nu mai este definit pointerul implicit *this*. Acest lucru conduce la faptul ca o functie prieten are un parametru in plus fata de o functie membru care are acelasi efect.

O functie prieten poate fi o functie obisnuita (ca in cazul functiei *modul* din exemplu de mai sus) sau o functie membru a unei alte clase.

Sa se defineasca tipul abstract de date complex care sa aiba functii membru pentru:

- modul, argument

- accesul la partea reala si imaginara a obiectelor de tip complex

- afisarea obiectelor complexe

- constructori pentru initializare si copiere

Sa se scrie un program care realizeaza urmatoarele:

- citeste perechi de n numere care reprezinta fiecare, partea reala si respectiv imaginara a unui numar complex

- afiseaza:

numarul complex citit

suma numerelor complexe citite

```
#include<iostream.h>
#include<math.h>
#define PI 3.14159265358979
class complex
{
// date membru protejate(private)
double real;
double imag;
// functii membru neprotejate
public:
```

```
// constructor folosit la initializare
complex(double x=0,double y=0);
// constructor de copiere
complex(const complex &);
// modulul numarului complex
double modul();
// argumentul numarului complex
double arg();
// returneaza partea reala
double retreal();
// returneaza partea imaginara
double retimag();
// afiseaza numarul complex
void afiscomplex();
};
complex::complex(double x, double y)
{
real=x;
imag=y;
}
complex::complex(const complex &z)
{
real=z.real;
imag=z.imag;
}
double complex::modul()
{
return sqrt(real*real+imag*imag);
```

```
}
double complex::arg()
{
double a;
if( (real==0) && (imag==0) ) return 0.0;
if(imag==0)
if(real>0) return 0.0;
else return PI;
if(real==0)
if(imag>0) return PI/2;
else return (3*PI)/2;
a=atan(imag/real);
if(real<0) return a+PI;
else
if(imag<0) return 2*PI+a;
return a; // real>0 si imag>0
}
double complex::retreal()
{
return real;
}
double complex::retimag()
{
return imag;
}
void complex::afiscomplex()
{
cout<<real<<" + i * "<<imag;
```

```
}
void main(void)
{
double x,y,sumax=0,sumay=0,i=0,n;
cout<<"Dati numarul de nr. complexe n = ";
cin>>n;
while(i<n)
{
cout<<"Partea reala = ";cin>>x;
cout<<"Partea imaginara = ";cin>>y;
sumax+=x;sumay+=y;
// se construieste nr. complex avand partile citite mai sus; se apeleaza constructorul care realizeaza initializarea
// obiectului z=x+i*y
complex z(x,y);
// afisez numarul z
cout<<endl;
z.afiscomplex();
i++;
}
complex sz=complex(sumax,sumay);
cout<<endl<<"Suma nr. complexe citite este ";
sz.afiscomplex();
}
```

Sa se defineasca o functie care ridica un numar complex la o putere intreaga pozitiva.

Un numar complex se poate ridica la o putere intreaga folosind formula lui **MOIVRE**. Aceasta se exprima prin relatia:

$$((r*\cos(a)+i*\sin(a)))^n = r^n * (\cos(n*a)+i*\sin(n*a))$$

unde:

r - este modulul numarului complex

a – este argumentul nr. complex

4.3. Modalităţi de supraîncărcare a operatorilor

O *funcţie operator* supraîncărcată poate fi introdusă ca *funcţie membră* (în general nestatică) sau *funcţie prieten* (funcţie *nemembră*).

Declararea unei funcţii membră nestatică specifică următoarele:

1. funcţia poate accesa partea privată a declaraţiei clasei

2. funcţia este în domeniul clasei

3. funcţia trebuie apelată dintr-un obiect

O funcţie membră statică satisface numai primele două condiţii, în timp ce o funcţie prietenă a unei clase satisface numai prima condiţie.

O funcţie operator având ca prim argument un tip primitiv nu poate fi funcţie membru.

Funcţiile care modifică reprezentarea unui obiect necesită acces la datele membri, deci trebuie să aparţină clasei. Ele sunt funcţii membrecu

argumente referinţe neconstante.

Dacă se doresc *conversii implicite* pentru termenii funcţiei operator, aceasta va fi funcţie nemembră cu argumente referinţe constante. Aceste funcţii implementează operatorii care nu necesită operanzi L-valori (adică se aplică tipurilor fundamentale). Aceştia sunt în general operatorii binari. Necesitatea accesului la reprezentare determină definirea lor ca funcţii prieten.

Operatorii care produc o nouă valoare din valorile argumentelor (de exemplu operatorul+) se definesc în afara clasei.

Dacă nu sunt necesare conversii de tip, alegerea între funcţie membru şi funcţie prieten rămâne la latitudinea programatorului.

O funcţie nemembră foloseşte numai argumente explicite, în timp ce o funcţie membră foloseşte argumentul implicit **this**.

Argumentele clase, transmise prin valoare se copiază în stivă neeconomic. Se preferă în acest caz argumentele referinţe constante.

Dacă *valoarea întoarsă de funcţie este o referinţă,* atunci aceasta *nu poate fi o variabilă automatică* (locală funcţiei). Ea *nu poate fi nici o variabilă statică locală*, deoarece operatorul poate fi folosit de mai multe ori într-o expresie. *Valoarea de întoarcere trebuie să fie alocată în memoria liberă (heap) sau să fie preluată dintr-un buffer de obiecte statice*.

Întoarcerea unei referinţe la obiectul modificat se realizează prin **return *this**.

Se caută minimizarea numărului de funcţii care au acces la reprezentarea internă a unei clase, prevăzându-se funcţii de acces.

În mod obligatoriu sunt funcţii membre: constructorii, destructorii, funcţiile virtuale, etc.

4.4. Operatori supraîncărcaţi ca funcţii prieten

Operatorii folosiţi în mod uzual pot fi unari sau binari. Utilizarea unui operator binar sub forma **a#b** este interpretată ca **operator#(a,b)**

Aşadar, un operator binar va fi reprezentat printr-o funcţie nemembră cu două argumente, iar un operator unar, printr-o funcţie nemembră cu un singur argument.

Argumentele se iau clase sau referinţe constante la clase (pentru o preluare economică, asigurând protecţia datelor)

Vom exemplifica pentru clasa **Cplx**:

```
class Cplx{
double re, im;
public:
Cplx(double x=0, double y=0);
Cplx(const Cplx& z);
//operatori binari
friend Cplx operator+(const Cplx& s, const Cplx& d);
friend Cplx operator-(const Cplx& s, const Cplx& d);
friend Cplx operator*(const Cplx& s, const Cplx& d);
friend Cplx operator/(const Cplx& s, const Cplx& d);
//operatori de comparatie
friend int operator==(const Cplx& s, const Cplx& d);
friend int operator!=(const Cplx& s, const Cplx& d);
//operatori unari
friend Cplx operator-(const Cplx& z);
friend Cplx operator!(const Cplx& z); //conjugat
friend Cplx& operator++(Cplx& z); //prefix
friend Cplx operator—-(Cplx& z,int); //postfix
};
```

```
//definitii operatori în afara domeniului clasei
Cplx operator+(const Cplx& s, const Cplx& d){
return Cplx(s.re+d.re,s.im+d.im);
};
Cplx operator-(const Cplx& s, const Cplx& d){
return Cplx(s.re-d.re,s.im-d.im);
};
Cplx operator*(const Cplx& s, const Cplx& d){
return Cplx(s.re*d.re-s.im*d.im,s.re*d.im+s.im*d.re);
};
Cplx operator/(const Cplx& s, const Cplx& d){
double t=d.re*d.re+d.im*d.im;
return Cplx((s.re*d.re+s.im*d.im)/t ,
(s.im*d.re-s.re*d.im)/t);
};
int operator==(const Cplx& s, const Cplx& d){
return s.re==d.re && s.im==d.im;
};
int operator!=(const Cplx& s, const Cplx& d){
return s.re!=d.re || s.im!=d.im;
};
Cplx operator-(const Cplx& z){
return Cplx(-z.re, -z.im);
};
Cplx operator!(const Cplx& z){
return Cplx(z.re, -z.im);
};
Cplx& operator++(Cplx& z){ //prefix
```

```
return Cplx(z.re+1, z.im);
};
Cplx operator++(Cplx& z,int){ //postfix
Cplx t(z);
z.re+=1;
return t;
};
```

Pentru a face deosebirea între semnăturile funcţiilor operatori de incrementare prefix şi postfix s-a introdus un argument fictiv pentru cel din urmă.

Incrementarea prefix întoarce valoarea incrementată, deci trebuie să fie o L-valoare (rezultatul întors va fi o referinţă), în timp ce postincrementarea întoarce valoarea dinaintea incrementării.

4.5. Operatori supraîncărcaţi ca funcţii membri

Funcţiilor membru li se transmite un argument implicit (ascuns) **this** (adresa obiectului, care poate reprezenta primul termen), motiv pentru care un operator binar poate fi implementat printr-o funcţie membru nestatică cu un singur argument (termenul din dreapta operatorului). **a#b** este interpretat ca **a.operator#(b)**

O funcţie membru operator unar ca o funcţie membru nestatică fără argumente (**#a** se interpretează ca **a.operator#();** pentru operatorul postfixat **a#** convenţia este **a.operator#(int)**).

În acest context, operatorii clasei **Cplx** se scriu:

```
Class Cplx{
double re, im;
public:
Cplx operator+(const Cplx& d);
Cplx operator-(const Cplx& d);
Cplx operator*(const Cplx& d);
Cplx operator/(const Cplx& d);
Cplx& operator+=(const Cplx& d);
Cplx& operator-=(const Cplx& d);
Cplx& operator*=(const Cplx& d);
Cplx& operator/=(const Cplx& d);
int operator==(const Cplx& d);
int operator!=(const Cplx& d);
Cplx& operator-();
Cplx& operator!();
Cplx& operator++();
Cplx operator++(int);
};


Cplx Cplx::operator+(const Cplx& d){
return Cplx(re+d.re, im+d.im);
};
Cplx Cplx::operator-(const Cplx& d){
return Cplx(re-d.re, im-d.im);
};
Cplx Cplx::operator*(const Cplx& d){
return Cplx(re*d.re-im*d.im, re*d.im+im*d.re);
```

```
};
Cplx Cplx::operator/(const Cplx& d){
double t=d.re*d.re+d.im*d.im;
return Cplx((re*d.re+im*d.im)/t, (im*d.re-re*d.im)/t);
};
Cplx& Cplx::operator+=(const Cplx& d){
if(this!=&d){
re+=d.re;
im+=d.im;
};
return *this;
};
Cplx& Cplx::operator-=(const Cplx& d){
if(this!=&d){
re-=d.re;
im-=d.im;
};
return *this;
};
Cplx& Cplx::operator*=(const Cplx& d){
if(this!=&d){
double t=re;
re=re*d.re-im*d.im;
im=t*d.im+im*d.re;
};
return *this;
};
Cplx& Cplx::operator/=(const Cplx& d){
```

```
if(this!=&d){
double t1=re, t2=d.re*d.re+d.im*d.im;
re=(re*d.re+im*d.im)/t2;
im=(im*d.re-t1*d.im)/t2;
};
return *this;
};
int Cplx::operator==(const Cplx& d){
return re==d.re && im==d.im;
};
int Cplx::operator!=(const Cplx& d){
return re!=d.re || im!=d.im;
};
Cplx& Cplx::operator-(){
re=-re;
im=-im;
return *this;
};
Cplx& Cplx::operator!(){
im=-im;
return *this;
};
Cplx& Cplx::operator++(){
re++;
return *this;
};
Cplx Cplx::operator++(int){
Cplx t(re,im);
```

```
re++;
return t;
};
```

4.6. Supraîncărcarea operatorului de atribuire

Operaţia de atribuire simplă, dacă nu este supraîncărcată, realizează o copiere membru cu membru.

Pentru obiectele care nu conţin date alocate dinamic la iniţializare, atribuirea prin copiere membru cu membru funcţionează corect, motiv pentru care nu se supraîncarcă operatorul de atribuire.

Pentru clasele ce conţin date alocate dinamic, copierea membru cu membru, executată în mod implicit la atribuire conduce la copierea pointerilor la datele alocate dinamic, în loc de a copia datele.

Operatorul de atribuire poate fi redefinit numai ca funcţie membră, el fiind legat de obiectul din stânga operatorului =, o L-valoare, motiv pentru care va întoarce o referinţă la obiect.

O primă cerinţă la scrierea funcţiei **operator=** este evitarea autoatribuirii **a=a**. În acest scop se efectuează testul **this != &d** (unde **d** este parametrul funcţiei -obiectul din dreapta operatorului = .

Urmează apoi "*curăţirea*" membrului stâng (eliberarea memoriei alocate dinamic). Copierea propriu zisă este făcută de obicei folosind constructorul de copiere.

Funcţia operator returnează o referinţă la obiectul modificat (**return *this**).

Exemplificăm pentru clasa **String**, definită astfel:

```
class String{
char* s;
int n;
public:
String(); //c(onstruc)tor implicit
String(const char* p); //ctor de initializare
String(const String& r); //cr de copiere
~String(); //dtor
String& operator=(const String& d);
String& operator=(const char* p);
};
//ctor de copiere
String::String& operator=(const String& d){
if(this != &d){ //evitare autoatribuire
if(s) //curatire
delete [] s;
n=d.n; //copiere
s=new char[n];
strncpy(s, d.s, n);
};
return *this; //intoarce obiectul modificat
};
//ctor de initializare (cu un sir tip C)
String::String& operator=(const char* p){
if(s)
delete [] s;
n=strlen(p);
s=new char[n];
```

```
strncpy(s, p, n);
return *this;
};
```

4.7. Supraîncărcarea operatorului de indexare

Operatorul de indexare [] este binar si are forma:

expresie_1[expresie_2]

Functia **operator[]** trebuie sa fie membra a clasei, sa fie nestatica si sa aiba forma:

x[n] sau **x.operator[ ] (n)**

Al doilea operand, care are rolul indexului, poate fi la redefinire de orice tip.

Programul urmator ilustreaza un mod de uitilizare a operatorului []. Ideea programului este aceea de a crea un tablou de inregistrari care contin

informatii despre persoane. Se doreste accesul la inregistrari cat mai usor posibil, in mai multe moduri:

- dupa nume,

- dupa greutate

- dupa numarul de ordine.

In prima cautare, pentru a face cat mai usor accesul la inregistrari dupa numele persoanelor, se va face redefinirea operatorului de indexare [] avand ca parametru un sir de caractere care indica numele persoanei cautate.

In al doilea caz, pentru a face cat mai usor accesul la inregistrari dupa greutatea persoanei, vom face redefinirea operatorului de indexare avand ca paramentru de tip float care indica greutatea persoanei cautate.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class analize;
class pers
{
// datele personale
char nume[35];
double greutate;
int varsta;
friend analize;
public:
// functii de introducere persoane si tiparire
void init(char *s, double gr, int v);
void tipar();
};
```

```
void pers::init(char *s, double gr, int v)
{
strcpy(nume,s);
greutate=gr;
varsta=v;
}
void pers::tipar()
{
if(this) cout<<endl<<"Persoana: "<<nume<<"\tGreutatea : "
<<greutate<<"\tVarsta : "<<varsta;
else cout<<"Nu exista"<<endl;
}
// definim clasa care contine analizele mai multor persoane
class analize
{
pers *sir;
int n;
public:
// definire constructori
analize(){ n=5; sir=new pers[5]; }
analize(int nr) { n=nr; cout<<n; sir=new pers[n]; }
// redefinire operatori
void introd();
pers *operator[] (char *);
pers *operator[] (double);
pers *operator[] (int);
};
// definirea unei functii de inregistrare a persoanelor
```

```
void analize::introd()
{
for(int i=0;i<n;i++)
{
cout<<endl;
cout<<"Persoana "<<i+1<<" : ";
cin>>sir[i].nume;
cout<<"Greutatea: ";
cin>>sir[i].greutate;
cout<<"Varsta: ";
cin>>sir[i].varsta;
}
}
//redefinire pentru a permite cautarea inregistrarilor dupa nume
pers *analize::operator[] (char *nume)
{
for(int i=0;i<n;i++)
if(strcmp(sir[i].nume,nume)==0) return &sir[i];
return NULL;
}
//redefinire pentru a permite cautarea inregistrarilor dupa greutate
pers *analize::operator[] (double g)
{
for(int i=0;i<n;i++)
if(sir[i].greutate==g) return &sir[i];
return NULL;
}
```

```
pers *analize::operator[] (int index)
{
if(index<=n) return &sir[index-1];
else return NULL;
}


void main(void)
{


char c;int nr;
cout<<"Cate analize efectuati ? ";cin>>nr;
analize t(nr); t.introd();
while(1)
{
cout<<endl<<"Optiunea [g]reutate, [n]ume, [i]ndex, [e]xit ? ";
cin>>c;
switch(c)
{
case 'g' : double g;
cout<<"Greutatea: ";cin>>g;
t[g]->tipar();
break;
case 'n' : char n[100];
cout<<"Numele: ";cin>>n;
t[n]->tipar();
```

```
break;
case 'i' : int i;
cout<<"Nr. de index: ";cin>>i;
t[i]->tipar();
break;
case 'e': return;
}
}
}
```

4.8. Supraîncărcarea operatorilor new şi delete

Functiile operator **new** si **delete** pot fi redefinite de catre utilizator cu conditia ca acestea sa fie membre ale clasei. Ele sunt in mod implicit de tip static.

Functia operator **new** are ca parametru tipul *size_t* si intoarce un pointer de tip void continand adresa zonei alocate. Ea are prototipul:

void *operator new(size_t);

Indicarea tipului *size_t* este obligatorie in C++ chiar daca tipul *size_t* este echivalent cu tipul *unsigned int*. Operatorul **new** supradefinit pastreaza proprietatea de a apela intotdeauna constructorul clasei dupa alocarea dinamica a obiectului. Calculul dimensiunii zonei de memorie si generarea obiectului raman in sarcina compilatorului.

Functia operator delete trebuie sa primeasca ca parametru un pointer de tipul clasei asociate sau pointer la void, continand adresa obiectului ce va

eliminat. Functia nu intoarce nici o valoare. Are prototipul:

void *operator delete(void *);

Operatorul delete supradefinit pastreaza proprietatea de a apela intotdeauna destructorul clasei inainte de a elibera memoria laocata dinamic obiectului.

Utilizarea versiunilor standard new si delete se poate face cu ajutorul operatorului de rezolutie (::).

Este posibila supradefinirea globala a operatorilor new si delete folosind functii independente. In acest caz, noii operatori vor fi apelati pentru toate tipurile de date, inclusiv datele standard. In aceasta situatie operatorii nu mai pot fi folositi in varianta lor standard si gestiunea alocarii dinamice revine in intregime programatorului.

Exemplu de redefinire a operatorilor new si delete:

```
#include<iostream.h>
class exemplu{
int a,b,c;
static short l;
public:
exemplu() { cout<<"Apel constructor exemplu "<<endl; }
~exemplu() { cout<<"Aple destructor exemplu "<<endl; }
void *operator new(size_t ln)
{
cout<<"Operator new redefinit ! ";
return (void *) new char[ln];
```

```
}

void *operator delete(void *p)
{
cout<<"Operator delete redefinit ! ";
delete[l] (char*)p;
}
};
```

Importanta practica a redefinirii operatorilor **new** si **delete** este sublinita in urmatoarele doua idei:

- Se pot gestiona mult mai bine alocarile, verificarile si setarile speciale in cazul datelor alocate prin intermediul lor;

- O utilizare speciala poate fi aceea in care dupa epuizarea memoriei speciale de tip *heap*, se poate face alocarea in alte zone de memorie (un fisier de pe disc poate fi folosit ca memorie virtuala in acest caz).

4.9. Supraîncărcarea operatorului apel de funcţie

Cand este realizata supraincarcarea operatorului "**apel de functie**", aceasta nu va insemna o noua varianta de redefinire a unei functii, ci redefinirea unui operator binar nestatic de forma:

expresie (lista_expresii);

Mentionam specificul utilizarii acestei functii operator:

- evaluarea si verificarea listei de argumente se face intocmai ca pentru o functie obisnuita.

chiar daca operatorul este binar, functia operator poate avea oricate argumente.

- Acesta deoarece al doilea argument este considerat o lista de argumente. Atunci cand lista de parametrii este vida, al doilea rgument poate lipsi.

Acest exemplu foloseste **operatorul ()** pentru parcurgerea unei liste simple.

```
#include<iostream.h>
#include<stdlib.h>
struct elem{
int inf;
elem *adr;
// se initializeaza camp inf cu n
elem(int n, struct elem *p): inf(n),adr(p){}
};
class list{
elem *prim,*p;
public:
```

```
// functie de afisare element curent
void afispoz()
{ cout<<"Elementul de pe pozitia curenta are valoarea "<<p->inf<<endl;}
//primul element va avea adresa absoluta zero
list(void) {prim=NULL;};
//crearea listei care va avea un numar dat de elemente
list(int);
// redefinirea operatorului realizeaza pozitionarea la nodul urmator; Daca pozitia curenta este la sfarsitul listei, atunci nodul urmator va fi primul element al listei
elem *operator() ();
// redefinirea operatorului determina avansarea nodului curent cu un numar dat de pozitii
elem *operator() (int);
};
list::list(int nr)
{
int inf;
if(nr<=0)
{ cout<<"Eroare la initializarea listei!";exit(1);}
cout<<endl<<"Introduceti "<<nr<<" valori ";cin>>inf;
prim=new elem(inf,NULL);
elem*q=prim;
for(int i=1;i<nr;i++)
{
cin>>inf;
p=new elem(inf,NULL);
q->adr=p;
```

```
q=p;
}
p=prim;
}
elem *list::operator() ()
{
elem *poz=p;
poz=p->adr ? p->adr : prim;
return p=poz;
}


elem *list::operator() (int nr)
{
for(int i=1;i<nr;i++)
(*this) (); // se pozitioneaza cu nr-1 pozitii
return (*this)(); // valoarea returnata este data de ultima pozitionare
}
void main()
{
list l(6);
l();
l.afispoz();
l(2);
l.afispoz();
}
```

4.10. Supraîncarcarea altor operatori

Stim ca exista patru operatori (**. :: ?: .***) care nu pot fi redefiniti. In rest orice alt operator poate fi redefinit.

a) De exemplu ***operatorul virgula***, poate fi supraincarcat. Fiind in operator binar, pentru a face redefinirea sa trebuie ca functia operator sa intoarca adresa operatorului din dreapta lasand nemodificati ambii operanzi.

```
#include<iostream.h>
class rational{
long numarator,numitor;
public:
rational operator, (rational n);
rational (long a=0, long b=1) { numarator=a; numitor=b; }
void afisare() { cout<<" "<<numarator<<"/"<<numitor;}
};
rational rational::operator, (rational n)
{
rational temp=n;
```

```
cout<<endl<<"temp = ";
temp.afisare();
return temp;
}
void main(void)
{
rational a(3,5), b(5,7), c(7,9);
a.afisare();
b.afisare();
c.afisare();
a=(b,c); cout<<endl;
a.afisare();
b.afisare();
c.afisare();
}
```

In urma executiei programului se vor afisa urmatoarele rezultate:

3/5 5/7 7/9

temp 7/9

7/9 5/7 7/9

b) ***Operatorul de selectie indirecta*** "**->**" poate fi la randul sau supraincarcat, fiind considerat un operator unar. Deoarece modul sau de utilizare este:

obiect->camp

variabila obiect va apela functia operator, care trebuie sa intoarca un pointer la obiectul asupra caruia se apeleaza. Cand operatorul este redefinit ca functie membru sintaxa sa este:

operator->()

In exemplul urmator se redefineste operatorul "**->**" astfel incat acesta sa aiba acelasi efect ca operatorul "**.**" (*operatorul de selectie directa*).

```
class c{
public: int n;
c* operator->() { cout<<"Redefinire operator -> "; return this;} // . . . .};
```

De acum, in utilizarea unui obiect *ob* din clasa *c*, *ob->i* va echivalent cu *ob.i*.

Clasa Matrice folosita pentru operatii cu matrice de numere reale.

```
#include<iostream.h>
#include<iomanip.h>
#include<stdlib.h>
```

```
#include<assert.h>
#include<conio.h>
class matrice{
float **a; //pointer catre matrice
int l,c; // linie si coloana
public:
//constructor
matrice(int l1=1, int c1=1, float vi=0);
//constructor implicit
matrice(matrice &x);
//destructor
~matrice();
//supraincarcarea operatorului "="
matrice &operator=(matrice &x);
// valoarea unei matrice
float *val(int l1, int c1);
// returneaza numarul de linii ale matricii
int linii() const { return l; }
// returneaza numarul de coloane ale matricii
int coloane() const { return c; }
//
void setl(int l1) { l=l1; }
//
void setc(int c1) { c=c1; }
// supraincarcarea operatorului "+" printr-o functie prieten
friend matrice operator+(const matrice &x, const matrice &y);
// supraincarcarea operatorului "*" printr-o functie prieten
friend matrice operator*(const matrice &x, const matrice &y);
```

```
// supraincarcarea operatorului "==" printr-o functie prieten
friend int operator==(const matrice &x, const matrice &y);
// supraincarcarea operatorului "<<" printr-o functie prieten
friend ostream& operator<<(ostream& os, const matrice &y);
// supraincarcarea operatorului ">>" printr-o functie prieten
friend istream& operator<<(istream& os, matrice &y);
};
matrice::matrice(int l1, int c1, float vi)
{
l=l1;
c=c1;
a=new float* [l];
for(int i=0;i<l;i++)
{
a[i]= new float [c];
for(int j=0;j<c;j++)
a[i][j]=vi;
}
}
matrice::matrice(matrice& x)
{
l=x.linii();
c=x.coloane();
a=new float* [l];
for(int i=0;i<l;i++)
{
a[i]= new float [c];
for(int j=0;j<c;j++)
```

```
a[i][j]=x.val(i,j);
}
}
matrice::~matrice()
{
for(int i=0;i<l;i++)
delete [] a[i];
delete [] a;
}
matrice& matrice::operator=(matrice &x)
{
if(this==&x) return *this;
l=x.linii();
c=x.coloane();
a=new float* [l];
for(int i=0;i<l;i++)
{
a[i]= new float [c];
for(int j=0;j<c;j++)
a[i][j]=x.val(i,j);
}
return *this;
}
float &matrice::val(int l1, int c1)
{
assert(l1>=0 && l1<=l && c1>=0 && c1<=c);
return a[l1][c1];
}
```

```
matrice operator+(const matrice &x, const matrice &y)
{
assert(x.linii()==y.linii() && x.coloane()==y.coloane());
int lin=x.linii();
int col=x.coloane();
matrice z(lin,col);
for(int i=0;i<lin;i++)
for(int j=0;j<col;j++)
z.val(i,j) = x.val(i,j) + y.val(i,j);
return z;
}
matrice operator*(const matrice &x, const matrice &y)
{
assert(x.coloane()==y.linii());
int lz=x.linii();
int cz=y.coloane();
int col=x.coloane();
matrice z(lz,cz);
for(int i=0;i<lz;i++)
for(int j=0;j<cz;j++)
{
float s=0;
for(int k=0;k<col;k++) s+=x.val(i,k)*y(k,j);
z.val(i,j) = s;
}
return z;
}
int operator==(const matrice &x, const matrice &y)
```

```
{
if(x.linii()!=y.linii() || x.coloane()!=y.coloane()) return 0;
for(int i=0;i<x.linii();i++)
for(int j=0;j<x.coloane();j++)
if(x.val(i,j) != y.val(i,j)) return 0;
return 1;
}
ostream& operator<<(ostream &os, const matrice &y)
{
os<<endl;
for(int i=0;i<x.linii();i++)
{
for(int j=0;j<y.coloane();j++)
os<<y.val(i,j)<<" ";
os<<endl;
}
return os;
}
istream& operator>>(istream &is, matrice &y)
{
int l,c;
cout<<"Numar de linii: ";is>>l;y.setl(l);
cout<<"Numar de coloane: ";is>>c;y.setc(c);
cout<<"Elementele pe linii"<<endl;
for(int i=0;i<y.linii();i++)
for(int j=0;j<x.coloane();j++)
is>>y.val(i,j);
return is;
```

```
}
void main(void)
{
matrice a,b,c;
clrscr();
cin>>a;
cout<<a;
cin>>b;
cout<<b;
c=a+b;
cout<<c;
c=a*b;
cout<<c;
getch();
}
```

Clasa vector folosita pentru operatii cu vectori de numere intregi.

```
#include<iostream.h>
#include<conio.h>


class vector{
private:
int *a;
```

```
//vectorul va avea un numar variabil de componente si deci spatiul necesar il vom aloca in heap
int n;
public:
vector();// constructor implicit
vector(int *a, int n1);// constructor
~vector(); //destructor
int min(); // determina cel mai mic element din vector
void afisare();//afisarea elem. unui vector
friend int egali(vector &v1, vector &v2);// testeaza egalitatea a doi vectori
int operator<(vector &v);// compararea a doi vectori
void ord_cresc();// ordonarea crescatoare a elem. unui vector
friend void inter(vector &v1, vector &v2, vector &v3);// interclasarea a doi vectori
};
vector::vector(int *a1, int n1)
{
n=n1;
a=new int[n];
for(int i=0;i<n;i++) a[i]=a1[i];
}
vector::~vector()
{
cout<<"se elibereaza "<<n*sizeof(int)<<" octeti"<<endl;
delete[] a;
}
int vector::min()
{
```

```
int m;
m=a[0];
for(int i=1;i<n;i++)
if(a[i] < m) m=a[i];
return m;
}
void vector::afisare()
{
for(int i=0;i<n;i++) cout<<a[i]<<" ";
cout<<endl;
}
int egali(vector &v1, vector &v2)
{
if(v1.n!=v2.n) return 0;
for(int i=0;i<v1.n;i++)
if(v1.a[i]!=v2.a[i]) return 0;
return 1;
}
int vector::operator<(vector &v)
{
int k;
if(n < v.n) k=n;
else k=v.n;
for(int j=0;j<k;j++)
if(a[j]>v.a[j]) return 0;
else
if(a[j]<v.a[j]) return 1;
if(n < v.n) return 1;
```

```
else return 0;
}
void vector::ord_cresc()
{
//ordonarea elem. unui vector fol. metoda bubblesort
int i,j,aux,k;
do{
k=1;
for(i=0;i<=n-1;i++)
if(a[i]>a[i+1])
{
aux=a[i];
a[i]=a[i+1];
a[i+1]=aux;
k=0;
}
}while(k==0);
}
void inter(vector &v1, vector &v2, vector &v3)
{
int i=0,j=0,k=0,l;
while( (i<=v1.n) && (j<=v2.n) )
if(v1.a[i]<v2.a[j])
{
v3.a[k]=v1.a[i];
i++;k++;
}
else
```

```
{
v3.a[k]=v2.a[j];
j++;k++;
}
if(i<=v1.n)
for(l=i;l<v1.n;l++) v3.a[l++]=v1.a[l];
else
for(l=j;l<v2.n;l++) v3.a[l++]=v2.a[l];
}
void main(void)
{
clrscr();
int x[]={7,6,3,3,1};
int y[]={6,55,44,3,2,10};
vector v1(x,5), v2(y,6);


cout<<"Vectorul v1 = ";v1.afisare();
cout<<"Vectorul v2 = ";v2.afisare();


cout<<"Minimul din v1 este egal cu "<<v1.min()<<endl;
cout<<"Minimul din v2 este egal cu "<<v2.min()<<endl;


if(egali(v1,v2)) cout<<"v1 si v2 sunt egali"<<endl;
else cout<<"v1 si v2 sunt diferiti"<<endl;
if(v1 < v2) cout<<"v1 < v2"<<endl;
else cout<<"v1 nu este mai mic decat v2"<<endl;
```

```
cout<<endl;
cout<<"Vectorul v1 ordonat crescator : ";
v1.ord_cresc();v1.afisare();
cout<<"Vectorul v2 ordonat crescator : ";
v2.ord_cresc();v2.afisare();


int z[50];
vector v3(z,11);
cout<<"Vectorul interclasat v3 = ";inter(v1,v2,v3);
v3.afisare();
getch();
}
```

In urma executiei programului se vor afisa urmatoarele rezultate:

```
Vectorul v1 = 7 6 3 3 1
Vectorul v2 = 6 55 44 3 2 10
Minimul din v1 este egal cu 1
Minimul din v2 este egal cu 2
v1 si v2 sunt diferiti
v1 nu este mai mic decat v2

Vectorul v1 ordonat crescator : 1 3 3 6 7
Vectorul v2 ordonat crescator : 2 3 6 10 44 55
Vectorul interclasat v3 = 1 2 3 3 3 6 6 7 10 44 55
```

4.11. Aplicatii

Programul urmator introduce clasa **fractie** ca fiind formata din doi intregi cu semn; avand in vedere ca semnul nu poate fi decat + sau -, din considerente de tiparire, se stocheaza doar la numerator. Constructorul cu valori implicite creeaza fractia 0 / 1; cand constructorul primeste un numar oarecare, ii pune numitor 1, dupa ce in prealabil ii indeparteaza partea zecimala! Impartirea la zero se trateaza prin constructor simuland infinitul prin MAX_INT.

Doua functii utilitare specifice lucrului cu numere intregi, determina **cel mai mare divizor comun** dupa algoritmul lui Euclid si **cel mai mic multiplu comun**, ca raport intre produsul numerelor sic el mai mare divisor comun. Cele doua functii folosesc la aducerea la acelasi numitor comun a doua fractii.

Amplificarea si **simplificarea** sunt alte doua operatii uzuale, carora le corespunde cate o functie membra. Simplificarea se face cu cel mai mare divizor comun, in timp ce amplificarea primeste factorul cu care se opereaza.

Operatiile de baza reproduce varianta uzuala de lucru cu fractii.

Pentru a reduce efortul de programare, a fost introdusa supraincarcarea **operatorului unar de semn (-)**, iar scaderea a devenit astfel adunare cu opusul fractiei. Din aceleasi considerente, impartirea este descrisa ca inmultire cu inversul fractiei.

Pentru operatorii relationali, se prefera aducerea fractiilor la **double**, pentru comparatii simple intre valorile lor.

```
#include<iostream.h>
#include<math.h>
```

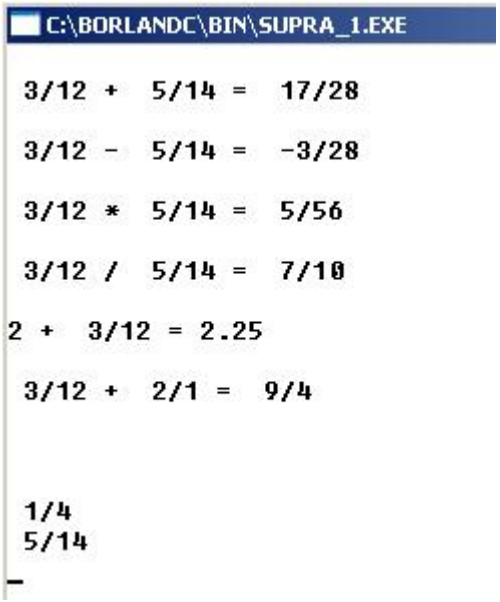
```
#include<conio.h>
#define MAX_INT 0x7FFFFFFF


int cmmdc(int a, int b)
{
// varianta recursiva
if(a%b) return cmmdc(b,a%b);
else return b;
}


int cmmmc(int a, int b)
{
return a*b/cmmdc(a,b);
}


class fractie{
int nr,nm; // nr = numarator; nm = numitor
public:
fractie(int n=0, int m=1)
{
if(m<0)
{
m=-m;
n=-n;
}
nr = (m ? n : MAX_INT);
```

```
nm = (m ? m : 1);
}
friend ostream &operator<<(ostream& ies, fractie& f);
void simplific();
void amplific(int factor);
fractie operator+(fractie f2);
fractie operator-();
fractie operator-(fractie f2);
fractie operator*(fractie f2);
fractie operator/(fractie f2);
operator double() { return (double) nr/nm; }
};
ostream &operator<<(ostream& ies, fractie &f)
{
ies<<" "<<f.nr<<"/"<<f.nm;
return ies;
}
void fractie::simplific()
{
int factor=cmmdc(abs(nr),abs(nm));
nr=nr/factor;
nm=nm/factor;
}
void fractie::amplific(int factor)
{
nr=nr*factor;
nm=nm*factor;
}
```

```
fractie fractie::operator+(fractie f2)
{
fractie f1=*this, rezultat;
f1.simplific();
f2.simplific();
int mc=cmmmc(f1.nm,f2.nm);
f1.amplific(mc/f1.nm);
f2.amplific(mc/f2/nm);
rezultat.nr = f1.nr + f2.nr;
rezultat.nm = mc;
rezultat.simplific();
return rezultat;
}
fractie fractie::operator-()
{
return fractie(-nr,nm);
}
fractie fractie::operator-(fractie f2)
{
return fractie(*this + (-f2));
}
fractie fractie::operator*(fractie f2)
{
fractie temp;
temp.nr = nr * f2.nr;
temp.nm = nm * f2.nm;
temp.simplific();
return temp;
```

```
}
fractie fractie::operator/(fractie f2)
{
return *this * fractie(f2.nm, f2.nr);
}
void main(void)
{
clrscr();
fractie f1(3,12), f2(5,14);
cout<<endl<<f1<<" + "<<f2<<" = "<<f1+f2<<endl;
cout<<endl<<f1<<" - "<<f2<<" = "<<f1-f2<<endl;
cout<<endl<<f1<<" * "<<f2<<" = "<<f1*f2<<endl;
cout<<endl<<f1<<" / "<<f2<<" = "<<f1/f2<<endl;
cout<<endl<<2<<" + "<<f1<<" = "<<2+f1<<endl;
cout<<endl<<f1<<" + "<<(fractie)2<<" = "<<f1+(fractie)2<<endl;
f1.simplific();
f2.simplific();
cout<<endl<<endl<<endl;
cout<<f1<<endl;
cout<<f2<<endl;
getch();
}
```

Rezultatele afisate de programul de mai sus:



Metodele definite pana acum sunt suficiente pentru cele mai uzuale operatii; chiar si adunarea unei fractii cu un numar este posibila, cand constructorul va intra automat in functie prin transformarea numarului in fractie.

Nu se pot efectua insa alte operatii intre un numar si o fractie; pentru acoperirea acestor situatii a fost introdus un operator ***cast*** spre **double**, care transforma astfel de operatii in simple operatii pe **double**.

Exemplul 2:

Programul urmator introduce clasa **BigInt** pentru prelucrarea numerelor intregi mai mari decat cel mai mare tip predefinit al limbajului C++ - **long**. Se stie ca tip long retine numere intregi aflate in intervalul [-2147483647, 2147483647], cu semn sau [0, 4294967295] fara semn. Pentru a lucra cu valori mai mari, programatorul trebuie sa preia el sarcinile unei aritmetici

extinse, astfel incat putem defini clasa **BigInt** care stocheaza numere intregi pozitive, de dimensiuni mari, sub forma unui sir de cifre.

Cifrele vor fi retinute intr-un vector de caractere numit **cifra[NR_CIF]**, unde **NR_CIF** este o constanta cu valoarea 100 reprezentand numarul maxim de cifre cu care poate lucra clasa **BigInt**. In acest vector de tip **char** cifrele vor fi retinute la nivel de reprezentare interna, nu de cod ASCII, in acest fel, la nivel de cifra calculele se pot face direct, fara nici o conversie prealabila. Ordinea de stocare este de la stanga la dreapta, adica cifrelor de rang mare le corespund elementele de inceput ale vectorului. Numarul cifrelor semnificative ale unui obiect se determina cu un apel al fucntiei membru numite **nr_cifre()**.

```
#include<iostream.h>
#include<string.h>
#include<conio.h>
#include<iomanip.h>
#define NR_CIF 30


class BigInt{
signed char cifra[NR_CIF];
public:
BigInt(long val=0);
BigInt(char *);


BigInt operator+(BigInt &);
BigInt operator+(int);
BigInt operator+(char *);
BigInt prod_1(signed char);
```

```
void deplasare(int);
int nr_cifre();
BigInt operator*(BigInt &);
friend ostream &operator<<(ostream&, BigInt&);
};


BigInt::BigInt(long val)
{
register i;
for(i=0;i<NR_CIF;i++) cifra[i]=0;
for(i=NR_CIF-1;val && i>=0; i--)
{
cifra[i]=val%10;
val/=10;
}
}


BigInt::BigInt(char *sir_cif)
{
register i,j;
for(i=0;i<NR_CIF;i++)
cifra[i]=0; // cifrele in exces sunt 0
for(i=NR_CIF-strlen(sir_cif), j=0; i<NR_CIF; i++, j++)
cifra[i]=sir_cif[j]-'0'; // cifrele obisnuite se retin in cod intern-ASCII
}
```

```
BigInt BigInt::operator+( BigInt &bi2)
{
BigInt temp;
int deplas=0, i;
for(i=NR_CIF-1; i>=0; i--)
{
temp.cifra[i]=cifra[i] + bi2.cifra[i] + deplas;
if(temp.cifra[i] > 9)
{
temp.cifra[i] %= 10;
deplas=1;
}
else
deplas=0;
}
return temp;
}


BigInt BigInt::operator+(int n)
{
return *this + BigInt(n);
}


BigInt BigInt::operator+(char *sir_cif)
{
return *this + BigInt(sir_cif);
```

```
}


BigInt BigInt::prod_1(signed char cif)
{
BigInt aux=*this;
char deplas=0, i;
for(i=NR_CIF-1; i>=0; i--)
{
aux.cifra[i]=aux.cifra[i] * cif + deplas;
if(aux.cifra[i] > 9)
{
deplas=aux.cifra[i]/10;
aux.cifra[i]%=10;
}
else deplas=0;
}
return aux;
}


void BigInt::deplasare(int n)
{
int i,j;
if(n>0 && n<NR_CIF)
{
for(i=0, j=n; j<NR_CIF; i++, j++)
cifra[i]=cifra[j];
```

```
for( ; i<NR_CIF; i++)
cifra[i]=0;
}
}


int BigInt::nr_cifre()
{
for(int i=0; cifra[i]==0;i++) ;
return NR_CIF-i;
}


BigInt BigInt::operator*(BigInt &b2)
{
int nr_cif, nr_poz, i;
BigInt rez;
nr_cif=b2.nr_cifre();
for(i=NR_CIF-1, nr_poz=0; i>=NR_CIF-nr_cif; i--)
{
BigInt aux=prod_1(b2.cifra[i]);
aux.deplasare(nr_poz);
cout<<"\nxx"<<setw(50-aux.nr_cifre())<<" "<<aux;
nr_poz++;
rez = rez + aux;
}
return rez;
}
```

```
ostream &operator<<(ostream& out, BigInt &bi)
{
register i;
for(i=0; (bi.cifra[i]==0) && i<NR_CIF; i++)
if(i == NR_CIF)
out<<0;
else
for( ; i<NR_CIF; i++)
out<< (char)(bi.cifra[i] + '0');
return out;
}
void main(void)
{
clrscr();
BigInt b1(12345678901234567890), b2(9999);
cout<<"\nb1 = "<<b1<<endl;
cout<<"\n"<<b1<<" + "<<b2<<" = "<<b1+b2<<endl;


b1="123456789";
b2="1999999";
cout<<"\n"<<b1<<" * "<<b2<<" = "<<b1*b2<<endl;
getch();
}
```

Constructorii de clasa genereaza obiecte pornind de la numere intregi obisnuite (implicit 0) sau de la numere date ca text, cu maxim **NR_CIF** cifre.

Calculele reproduc varianta manuala de lucru, adica operand cifra cu cifra si tinand minte cifra de transport.

Adunarea introdusa prin **operator+** recunoaste in intrare doua numere intregi mari, un numar intreg mare si unul mic, sau un numar intreg mare si altul dat ca text. Din supraincarcari se observa ca toate adunarile sunt pana la urma redirectionate cu ajutorul costructorilor catre adunarea a doua obiecte **BigInt**. O parte din redirectionari se faceau si implicit deoarece constructorii cu un parametru sunt folositi de compilator drept convertori, pentru a incerca identificarea unui prototip pentru **operator+**.

Operatia de inmultire se realizeaza in doua etape: **prod_1()** face inmultirea unui obiect **BigInt** cu o cifra si obtine tot un obiect **BigInt**; operatia se repeta pentru fiecare cifra a inmultitorului, obiectele auxiliare rezultate fiind deplasate corespunzator spre stanga (functia **deplasare()**) si adunate la rezultatul final. Pentru facilitarea intelegerii in supraincarcarea **operator***, a fost pusa o afisare intermediara.

Capitolul 5 - Clase derivate. Moştenire.

5.1. Noţiuni de bază care privesc moştenirea

Prin mostenire se intelege acea proprietate a claselor prin care un tip nou construit poate prelua datele si metodele unui tip mai vechi.

In C++ acest mecanism mai este cunoscut sub numele **derivarea claselor**.

Evident, clasei nou construite i se pot adauga noi date si metode.

Prin acest procedeu se preia soft deja facut care se dezvolta. Avantajul urias al acestui procedeu este ca persoana acre reia un anumit soft trebuie sa cunoasca doar documentatia de utilizare a soft-ului preluat.

Ideea de baza este urmatoarea: fiind date clasele X1, X2, …., Xn, putem construi clasa Xn+1 care are, pe langa datele si metodele claselor enumerate, date si metode proprii. Clasele X1, X2, …., Xn se numesc **clase de baza**, iar clasa Xn+1 **clasa derivata**.

Clasa **numar** contine o data membru n, si o metoda membru **tip()**. Clasa **numar_d** este derivata din clasa **numar**. Prin urmare, un obiect obtinut prin instantierea acesteia, are data membru n si metoda **tip()**. Pe langa aceasta, clasa **numar_d** are o noua metoda **tip()**.

```
#include<iostream.h>
```

```
class numar{
public:
int n;
void tip() { cout<<n<<endl; }
};


class numar_d : public numar{
public:
void cit()
{
cout<<"n = ";
cin>>n;
}
};


void main(void)
{
numar_d a;
a.cit();
a.tip();
}
```

Clasa **complex** ne permite sa lucram usor cu numere complexe. In timpul utilizarii acestei clase am observat a ar fi bine sa contina o metoda booleana

numita **real()**, care sa ne permita sa testam daca numarul complex este real sau nu. Dispunem de documentatia clasei **complex**. Vom defini o noua clasa numita **complex_d** care sa contina datele si metodele clasei complex si, in plus, sa contina si o noua metoda.

```
#include<iostream.h>
#include "complex.cpp"
class complex_d : public complex{
public:
int real();
};


int complex_d::real()
{
return y==0;
}


void main(void)
{
complex_d z;
z.x=3;
z.y=1;
if(z.real()) cout<<"numarul este real";
else cout<<"numarul nu este real";
}
```

Atribuiri intre obiectele unei clase derivate si obiectele clasei de baza

Unui obiect al clasei de baza i se poate atribui un obiect al unei clase care este derivata din ea.

Unui obiect al unei clase derivate nu i se poate atribui un obiect al clasei de baza.

De ce? Pentru ca in clasa derivata avem date si metode in plus fata de clasa de baza! si nu putem sti ce valori vor primi acestea in cazul unei astfel de atribuiri.

Putem totusi supraincarca **operatorul "="** in clasa derivata. Dar si aici datele care apartin exclusive clasei derivate vor lua valori arbitrare (in exemplul urmator valoarea 0).

```
#include<iostream.h>
class numar{
public:
int m;
};


class numar_d : public numar{
public:
int n;
void operator=(numar &x)
{
```

```
m=x.m;
n=0;
}
};


void main(void)
{
numar a;
numar_d b;
b.m=3;
b.n=4;
a=b;
cout<<a.m<<endl;
b=a; // nu arfi permisa
cout<<b.m<<" "<<b.n<<endl;
}
```

5.3. Supraincarcarea functiilor

Se stie ca functiile pot fi supraincarcate. Dar o clasa derivate poate contine o functie cu acelasi nume si cu aceeasi lista de parametrii formali ca una a clasei de baza?

Raspunsul este afirmativ. In exemplul urmator clasa de baza **numar** are o metoda numita **tip()**, in care lista parametrilor este vida, iar clasa derivata **numar_d** are pe langa metoda **tip()**, mostenita, o alta metoda numita tot **tip()**, tot cu lista parametrilor vida.

Prin **a.tip()** se apeleaza metoda **tip()** proprie clasei derivate. In cazul in care dorim sa apelam metoda **tip()** a clasei de baza scriem: **a.numar::tip()**. Cele doua metode cu acelasi nume difera doar prin mesajul afisat inaintea tiparirii valorii retinute de variabila.

```
#include<iostream.h>


class numar{
public:
int i;
void tip()
{
cout<<"numar "<<i<<endl;
}
};


class numar_d : public numar{
public:
void tip()
{
cout<<"numar_d "<<i<<endl;
}
};
```

```
void main(void)
{
numar_d a;
a.i=3;
a.tip();
a.numar::tip();
}
```

Mai exista situatia in care in clasa derivate sa definim o data membru cu acelasi nume cu o data din clasa de baza.

Nu se semnaleaza eroare in acest caza. Iar un obiect rezultata ca instantiere a unei clase derivate va avea doua date cu acelasi nume. Implicit se adreseaza variabila definite in clasa derivata, dar este posibil sa adresam si data din clasa de baza la fel ca si in cazul functiilor.

```
#include<iostream.h>
class numar{
public:
int i;
};

class numar_d : public numar{
public:
```

```
int i;
};


void main(void)
{
numar_d a;
a.i=2;
a.numar::i=3;
cout<<a.i<<" "<<a.numar::i;
}
```

5.4. Accesul la membrii unei clase derivate

Membrii unei clase (date si metode) sunt de trei feluri:

1. Publici – precedati de cuvantul cheie **public:** Acestia pot si accesati din exteriorul clasei.

2. Privati –precedati de cuvantul cheie **private:** Stim ca, implicit membrii unei clase sunt ptivati. Membrii private ai unei clase nu pot fi accesati din exteriorul clasei, decat din interiorul clasei respective.

3. Protejati – precedati de cuvantul cheie **protected:** Pana acum nu am folosit acest tip de membri. Rolul lor apare in momentul in care calasa este derivate. ei nu pot si accesati din exterior (la fel ca si membrii privati), dar pot fi accesati din clasele derivate.

Exista trei modalitati in care o clasa poate fi mostenita(derivata):

1. class numar_d: ***public*** numar
{
. . .
};
2. class numar_d: ***private*** numar
{
. . .
};
3. class numar_d: ***protected*** numar
{
. . .
};

1. A fost mostenita clasa precedata de cuvantul cheie ***public:***

- membrii publici ai clasei de baza raman publici si in clasa derivata;
- membrii privati ai clasei de baza raman private si in clasa derivate;
- membrii protejati ai clasei de baza raman protejati si in clasa derivata.

2. A fost mostenita precedata de cuvantul cheie ***private:***

In aceasta situatie clasa derivate are toti membrii private. In schimb, in interiorul clasei derivate putem accesa membrii publicii si protejati ai clasei de baza.

3. A fost mostenita precedata de cuvantul cheie ***protected:***

In aceasta situatie, membrii publici si protejati ai clasei de baza sunt protejati pentru noua clasa, iar cei private ai clasei de baza raman private si pentru clasa derivate. din interiorul clasei derivate putem accesa membrii publici si privati ai clasei de baza.

Forma generala a unei clase derivate:

class nume : [public sau private sau protected] nume_1,

: [public sau private sau protected] nume_2,

. . .

: [public sau private sau protected] nume_n

{

. . .

};

5.5. Constructori si destructori

In cazul in care definim o clasa derivata a unei clase de baza, constructorii si destructorii au un comportament specific.

Clasa **numar_d** mosteneste clasa **numar**. Fiecare dintre ele este inzestrata cu un constructor. Cei doi constructori difera doar prin sirul afisat. La declararea unui obiect care instantiaza clasa derivata se tipareste “numar” apoi “numar_d”.

```
#include<iostream.h>
class numar{
public:
numar()
{ cout<<"numar "<<endl; }
};


class numar_d : public numar{
public:
numar_d()
{ cout<<"numar_d "<<endl; }
};
void main(void)
{
numar_d a;
}
```

Astfel se observa ca la declarea unui obiect se apeleleaza mai intai constructorul clasei de baza si apoi cel al clasei derivate.

Lucrurile se complica atunci cand in clasa de baza avem mai multi constructori, iar in clasa derivate la fel alti constructori specifici clasei respective.

Exemplul 7:

```
#include<iostream.h>
class numar{
public:
numar()
{ cout<<"numar "<<endl; }
numar(int k)
{ cout<<"numar "<<k<<endl; }
};


class numar_d : public numar{
public:
numar_d()
{ cout<<"numar_d "<<endl; }
numar_d(int k) : numar(k)
{ cout<<"numar_d "<<k<<endl; }
};


void main(void)
{
numar_d a,b(1);
}
```

Clasa **numar** are doi constructori, unul fara parametric, al doilea cu un paramentru implicit. In mod asemanator, clasa derivata, **numar_d** are, de asemenea, doi constructori, unul fara parametru si unul cu un paramentru implicit.

Un constructor al clasei derivate, mai intai apeleaza constructorul fara parametrii ai clasei de baza, apoi efectueaza propriile operatii. Apelul constructorului clasei de baza se face automat, fara ca programatorul sa scrie ceva. In cazul in care acesta doreste sa fie apelat un anumit constructor, altul

de cat cel fara parametric, se procedeaza ca in exemplul precedent, unde constructorul cu un parametru al clasei derivate apeleaza pe cel cu un parametru al clasei de baza:

```
numar_d (int k) : numar (k)
{ cout<<”numar_d ”<<k<<endl; }
```

Programul tipareste: **numar, numar_d, numar 1, numar_d 1**.

In ceea ce priveste destructorii, acestia se apeleaza in ordinea inverse in care au fost apelati constructorii.

```
#include<iostream.h>

class numar{
public:
numar()
{ cout<<"numar "<<endl; }
~numar()
{ cout<<"numar destructor"<<endl; }
};

class numar_d : public numar{
public:
```

```
numar_d()
{ cout<<"numar_d "<<endl; }
~numar_d()
{ cout<<"numar_d destructor"<<endl; }
};


void main(void)
{
numar_d b;
}
```

Programul va tipari: **numar, numar_d, numar_d destructor, numar destructor.**

5.6. Ierarhii de clase

O clasă derivată poate fi la rândul ei clasă de bază. De exemplu:

```
class Angajat{ . . .};
class Sef : public Angajat { . . .};
class Director : public Sef { . . .};
```

O asemenea ierarhie se reprezintă de obicei printr-un arbore, dar în cazul moştenirii multiple apare un graf orientat aciclic.

Pentru a fi folosită ca bază, o clasă trebuie să fie definită; simpla declarare a clasei este insuficientă.

Să considerăm o funcţie membru **void afisare() const;** în clasa de bază, pe care o vom *redefini* în clasele derivate, având *aceeaşi semnătură*:

```
class Angajat{
String nume;
double salariu;
public:
Angajat(const String& p, double s);
void afisare() const;
};
void Angajat::afisare() const
{
cout<<nume<< " salariu: "<<salariu<<endl;
};
Angajat::Angajat(const String& n, double s) : nume(n), salariu(s) { };
```

```
class Sef : public Angajat{
int sectie;
public :
Sef(const String& n, double s, int sec);
void afisare() const;
};
void Sef::afisare() const{
Angajat::afisare();
```

cout<<"Sef Sectie: "<<sectie<<endl;

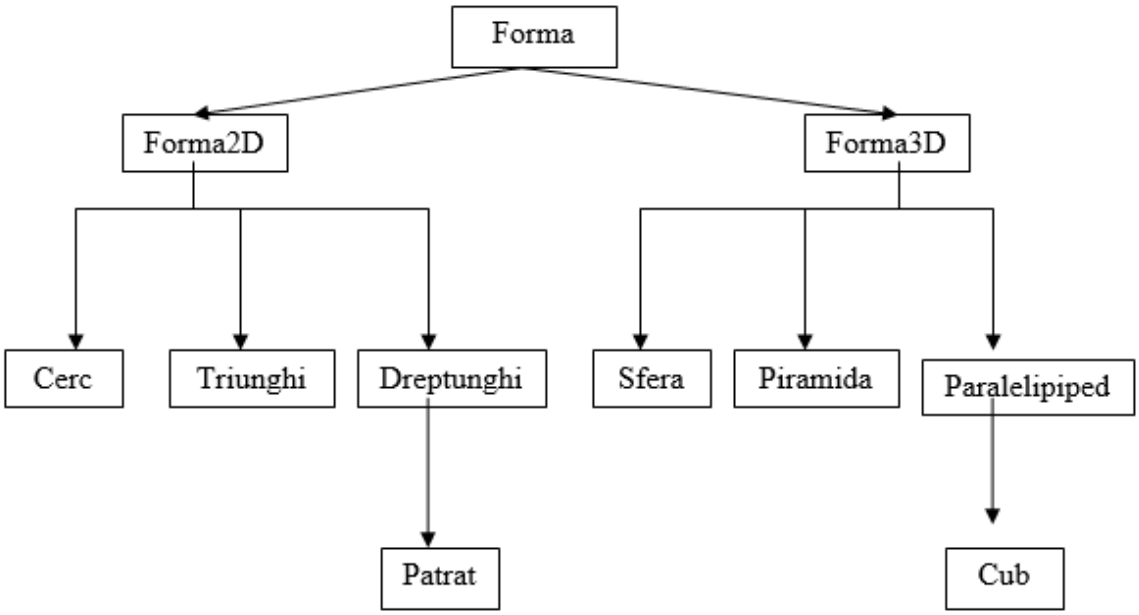
};

Sef::Sef(const String& n, doubles, int sec) : Angajat(n,s), sectie(sec) {};

Mostenirea genereaza o structurta ieharhica de tip arbore, o clasa de baza fiind intr-o relatie ierarhica cu clasa ei derivata.

Exemplu de *mostenire multipla*:

Consideram clasa de baza **Forma** care cuprinde toate formele posibile. Din aceasta putem deriva doua clase : **Forma2D** (care cuprinde formele bidimensionale) si **Forma3D** (care cuprinde formele tridimensionale). La randul lor acestea pot genera noi clase specializate, astfel clasa **Forma2D** poate genera clasele **Cerc**, **Triunghi**, **Dreptunghi**, iar clasa **Forma3D** poate genera clasele **Sfera**, **Piramida**, **Paralelipiped**. Prin relatia de mostenire multipla se obtine structua arborescenta, care este reprezentata grafic in figura urmatoare:



Exemple de *mosteniri singulare*:

Clasa de baza	Clase derivate
Persoana	Copil
	Elev
	Student
	Angajat
	Pensionar
Aparat_elecrocasnic	Frigider
	Aragaz
	Masina_de_spalat
Vehicul	Tren
	Avion
	Automobil

Constructori si destructori

Scopul claselor derivate este de a adauga noi caracteristici unei clase de baza. In general, clasele derivate au nevoie de constructori mai complecsi de cat cele de baza; de aceea, pentru usurinta, se vor apela implicit constructorii clasei de baza de cate ori este posibil. Deoarece o clasa de derivate nu este decat o “constructie” in jurul unei clase de baza, regulile sunt urmatoarele:

- pentru constructor, se apeleaza mai intai constructorii clasei de baza si apoi se executa actiunile specifice constructorului clasei derivate;

- pentru destructori, se procedeaza invers, apelandu-se inati destructorul clasei derivate si apoi cel al clasei de baza.

Daca toti constructorii sau destructorii unei clase de baza sunt de tip ***private***, atunci ei nu pot fi accesati de clasa derivate, rezultand o situatie grava, ce va fi semnalata ca eroare de compilare. Deci, se recomanda ca destructorul si cel putin un constructor al clasei de baza sa fie declarati de tip ***public***.

Constructorii si destructorii clasei de baza sunt folositi la crearea si distrugerea obiectelor claselor derivate, dar ei nu sunt mosteniti de catre acestea.

Exemplul 9:

```
#include<iostream.h>
#include<conio.h>


class BAZA{
protected:
int cont;
public:
void SET_CONTOR(int val)
{ cont=val; }
void TIP_CONTOR()
{ cout<<endl<<endl<<"CONTOR= "<<cont; }
BAZA (int a=0)
{
SET_CONTOR(a);
cout<<endl<<"Constructor BAZA;\t CONTOR = "<<cont<<endl;
}
~BAZA()
{
cout<<endl<<"Destructor BAZA;\t CONTOR = "<<cont<<endl;
}
};


class DERIVATA : public BAZA
{
public:
```

```
DERIVATA()
// apel implicit BAZA(0)
{ cout<<endl<<"Constructor DERIVATA "<<endl; }
DERIVATA(int b) : BAZA(b)
//apel explicit BAZA(b)
{ cout<<endl<<"Alt constructor DERIVATA "<<endl; }
void TIP_CONTOR()
// redefinirea functiei din clasa BAZA
{ cout<<endl<<endl<<"CONTOR DERIVATA = "<<cont; }
~DERIVATA()
// destructor
{ cout<<endl<<"Destructor DERIVATA"<<endl; }
};


void main(void)
{
clrscr();
BAZA b1, b2=1;
DERIVATA d1, d2(2), d3(3);


BAZA *pb1;
// la crearea pointerilor catre obiecte nu se apeleaza
// constructorii deoarece nu se cere initializarea obiectelor
DERIVATA *pd1;


pb1=&b2;
```

```
pd1=&d2;
pb1->TIP_CONTOR();
pd1->TIP_CONTOR();
cout<<endl;
getch();
}
```

La executia programului se vor afisa pe ecran mesajele urmatoare care arata clar ordinea de apel a constructorilor si destructorilor. Trebuie mentionat ca destructorii au fost apelati datorita incheierii blocului ce reprezinta domeniul de existenta al obiectelor respective.

```
Constructor BAZA; CONTOR = 0
Constructor BAZA; CONTOR = 1
Constructor BAZA; CONTOR = 0
Constructor DERIVATA
Constructor BAZA; CONTOR = 2
Alt constructor DERIVATA
Constructor BAZA; CONTOR = 3
Alt constructor DERIVATA


CONTOR = 1
CONTOR DERIVATA = 2


Destructor DERIVATA
Destructor BAZA; CONTOR = 3
```

Destructor DERIVATA

Destructor BAZA; CONTOR = 2

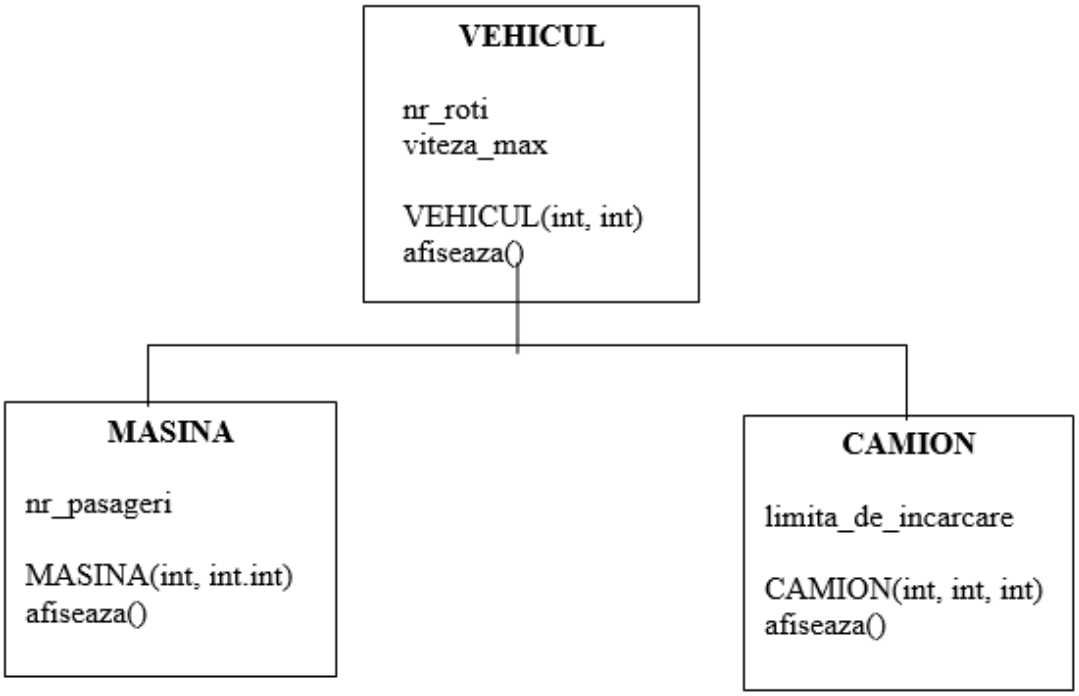
Destructor DERIVATA

Destructor BAZA; CONTOR = 0

Destructor BAZA; CONTOR = 1

Destructor BAZA; CONTOR = 0

Un alt exemplu privind utilizarea constructorilor cu lista de paramaterii in contextual mostenirii este descries prin modelul obiectual de mai jos.

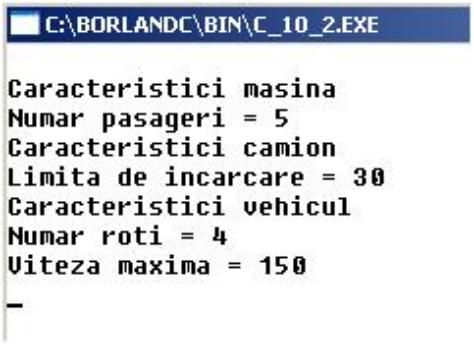


Exemplu 10:

```
#include<iostream.h>
#include<conio.h>


class VEHICUL{
int nr_roti;
int viteza_max;
public:
VEHICUL(int r, int v)
{ nr_roti=r; viteza_max=v; }
void afiseaza()
{
cout<<endl<<"Numar roti =
"<<nr_roti<<endl<<"Viteza maxima
= "<<viteza_max;
}
};


class MASINA : public VEHICUL
{
int nr_pasageri;
public:
MASINA(int p, int r, int v) : VEHICUL(r,v),
nr_pasageri(p) {}
void afiseaza()
{
```

```
cout<<endl<<"Numar pasageri =
"<<nr_pasageri;
}
};
class CAMION : VEHICUL
{
int limita_de_incarcare;
public:
CAMION(int l, int r, int v) :
limita_de_incarcare(l), VEHICUL(r,v) {}
void afiseaza()
{
cout<<endl<<"Limita de incarcare "
<<limita_de_incarcare;
}
};


void main(void)
{
clrscr();
MASINA M(5,4,150);
CAMION C(30,12,120);


cout<<endl<<"Caracteristici masina ";
M.afiseaza();
```

```
cout<<endl<<"Caracteristici camion ";
C.afiseaza();


cout<<endl<<"Caracteristici vehicul ";
M.VEHICUL::afiseaza();


cout<<endl;
getch();
}
```



Capitolul 6 - Polimorfism. Clase virtuale.

Polimorfismul este legat de mostenire.

Consideram o **clasa O1** care contine o **metoda M1** care o apeleaza pe alta **M2** din aceeasi clasa. O alta **clasa O2** o mosteneste pe prima. In cadrul acesteia redefinim **metoda M2**, dar nu redefinim **metoda M1**. Declaram un obiect de tip **O2** si apelam **metoda M1**. Problema este urmatoarea : care va fi metoda apelata de **M1**, **M2** din **O1** sau **M2** din **O2** ?

```
class O1
{
metoda M2 {….}
metoda M1 { metoda M2 ; }
};

class O2: public O1
{
metoda M2 { redefinita }
};

void main()
```

```
{
O2 obiect;
Obiect.M1;
}
```

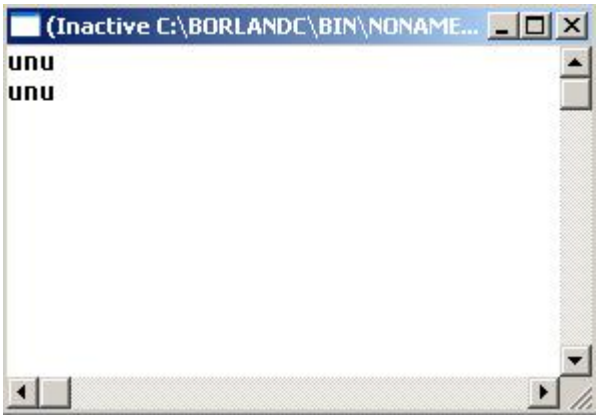
In situatia de mai sus va fi apelata metoda M2 din O1.

Consideram urmatorul modul care evidentiaza situatia precedenta : metoda **care()** apeleaza metoda **tip()**, redefinta.

```
#include<iostream.h>
class unu{
public :
void care()
{ tip() ; }
void tip()
{ cout<<"unu"<<endl; }
};

class doi : public unu{
void tip()
```

```
{ cout<<"doi"<<endl; }
};
void main()
{
unu x ;
doi y ;
x.care();
y.care();
}
```

Rezultatul executiei este:



In acest moment apare intrebarea: De ce am mai redefinit o functie (metoda) in noua clasa care mosteneste clasa de baza, daca nu putem folosi functia (metoda) redefinita si o apelam tot pe cea veche ?

O astfel de metoda de selectie a metodei prin obiectul care o apeleaza se numeste **selectia statica**. *Codul de apel al functiei este obtinut la compilare si ramane nemodificat pe parcursul executarii programului*.

Solutia rezolvarii unei astfel de probleme este sa se faca selectia in momentul apelului (si nu la compilare). In astfel de cazuri programatorul

trebuie sa decida momentul care metoda sa fie selectata. O astfel de selectie se numeste **selectie dinamica** sau **virtuala**.

Pentru a realiza selectia virtuala trebuie sa procedam astfel:

- cand se declara in cadrul clasei de baza, o metoda ce va redefinta, se adauga dupa declaratie cuvantul **virtual**;

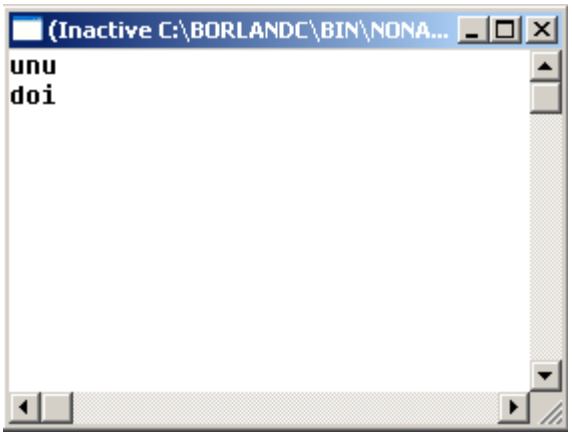
- orice redefinire a metodei, in cadrul claselor derivate, se va face utilizand acelasi cuvant cheie **virtual**;

Exemplul anterior se modifica corespunzator prin adaugarea cuvantului cheie virtual, atat in cadrul clasei de baza unu, cat si in cadrul clasei derivate doi. Rezultatul obtinut este cel asteptat:

```
#include<iostream.h>
class unu{
public :
void care()
{ tip() ; }
virtual void tip()
{ cout<<"unu"<<endl; }
};

class doi : public unu{
virtual void tip()
{ cout<<"doi"<<endl; }
```

```
};


void main()
{
unu x ;
doi y ;
x.care();
y.care();
}
```

Rezultatul executiei este:



Exemplul 3:

O aplicatie a polimorfismului

La cursul de ***Tehnici de programare*** am învăţat despre metoda backtracking standardizat. Am observat acolo ca metoda in sine consta dintr-o unica secventa de instructiuni care apela diferite functii care aveau intotdeauna acelasi nume, dar care se modificau de la un program la altul.

Vom incerca acum, sa construim o clasa in care functiile **init(), am_succesor()** si celelalte sa fie definite initial cu valori simple, iar apoi in clasele derivate sa poata fi redefinite astfel iincat sa rezolve fiecare problema in parte.

Dificultatea consta in faptul ca functia **run()** care contine algoritmul de backtracktring va apela functiile **init(), succesor()**, s.a.m.d. si deci obiectul care o foloseste trebuie sa contina si aceste functii. De aceea ele vor fi definite din start – in clasa de baza – ca fiind virtuale, pentru ca apoi in clasele derivate sa poata fi redefinite corespunzator fiecarei clase in parte care rezolva un anumit tip de problema. Deci, vom aplica polimorfismul

Initial in fisierul cu numele '**back.cpp**', vom declara clasa **bkt**, in care functiile **init(), am_succesor(), e_valid(), solutie() si tipar()** vor fi declarate virtuale. Deoarece ele nu vor fi niciodata apelate in cadrul acestei clase, vor avea 'corpul' vid. De asemenea, variabilele n,k, as, ev se gasesc in orice program backtracking si deci, vor fi declarate aici.

Fisierul **”back.cpp”** contine clasa **bkt** :

```
class bkt{
public:
int st[10],n,k;
virtual void init() {}
virtual int am_succesor() { return 0; }
virtual int e_valid() { return 0; }
virtual int solutie() { return 0; }
virtual void tipar() {}
void bkt::run();
};
void bkt::run()
{
int as;
k=1;init();
while(k>0)
{
do{
}while( (as=am_succesor()) && !e_valid() );
if(as)
if(solutie()) tipar();
else
{ k++; init(); }
else k--;
}
}
```

Exemple de probleme care pot si rezolvate acum prin utilizarea clasei **bkt**, prin polimorfism:

1. Generarea permutarilor

Se cere sa se genereze toate permutarile multimii {1,2,…,n}.

Pentru a folosi clasa **bkt** vom defini o alta clasa, care o mosteneste pe aceasta si care va avea functiile clasei initiale dar redefinite. Vom folosi un constructor care va avea rolul de a citi valoarea lui n.

Fisierul **”perm.cpp”** va defini clasa **permut** care va fi derivata din clasa **bkt**:

```
#include<iostream.h>
#include "back.cpp"

class permut:public bkt{
public:
permut (int v) { n=v; }
virtual void init();
virtual int am_succesor();
virtual int e_valid();
virtual int solutie();
```

```
virtual void tipar();
};


void permut::init()
{ st[k]=0; }


int permut::am_succesor()
{
if(st[k]<n){ st[k]++; return 1; }
else return 0;
}
int permut::e_valid()
{
for(int i=1;i<k;i++)
if(st[i]==st[k]) return 0;
return 1;
}
int permut::solutie()
{ return (k==n); }
void permut::tipar()
{
for(int i=1;i<=k;i++)
cout<<st[i]<<" ";
cout<<endl;
}
```

Se observa ca acest fisier – la fel ca si cel anterior – nu are functie main(), deoarece poate fi folosit in orice alt program care rezolva probleme de generarea permutarilor. Si anume poate fi folosit in urmatorul fiser care afiseaza rezultatul gerenerarii permutarilor multimii {1,2,…,n}:

```
#include<iostream.h>
#include<conio.h>
#include "perm.cpp"


void main()
{
clrscr();
permut x(3);
x.run();
getch();
}
```

2. Problema celor n regine(dame)

Se cere sa se genereze toate posibilitatile de asezare a n dame pe o tabla de sah de dimensiune n*n, astfel in cat sa nu se ‘atace’ reciproc.

Dupa cum stim, rezolvarea problemei difera de generarea permutarilor doar printr-o singura metoda: **e_valid()**. Atunci, vom redefini aceasta metoda din clasa **permut**, pentru a putea obtine solutia dorita:

Fisierul **”regine.cpp”** va contine clasa **dame** care va fi derivata din clasa **permut**:

```
#include<math.h>
#include "perm.cpp"


class dame : public permut{
public:
dame(int v) : permut(v) {};
virtual int dame::e_valid();
};


int dame::e_valid()
{
for(int i=1;i<k;i++)
if( (st[i]==st[k]) || abs(st[k]-st[i])==abs(k-i) ) return 0;
return 1;
}
```

Iar acum pentru scrierea programului care rezolva problem acelor n regine vom construi fisierul **“dame.cpp”** in care vom include fisierul **“regine.cpp”**:

Codul sursa al programului este:

```
#include<iostream.h>
#include<conio.h>
#include "regine.cpp"
```

```
void main()
{
clrscr();
dame x(8);
x.run();
getch();
}
```

Capitolul 7 - Funcţii Template (Şabloane)

7.1. Template-uri (template=şablon, model, tipar)

Template-urile C++ permit definirea de functii si clase care au parametri pentru numele de tipuri.

Exemplul 1:

```
void schimba (Tip_Var& var1,
Tip_Var& var2)
  {
      Tip_Var temp;

         temp=var1;
         var1=var2;
         var2=temp;
      }
```

unde Tip_Var poate fi **int**, **char**, **double** si altele.

Singura diferenta dintre aceste functii este ca difera tipul parametrilor formali.

Aceste functii se pot scrie intr-una singura in urmatorul program C++:

Program ce ilustreaza o functie template.

```
#include <iostream.h>


template <class T>
void schimba(T& var1, T& var2);
// interschimba valorile lui var1 si var2
int main()
{
int i1 = 1, i2 = 2;
cout << "Initial, i1 = " << i1 << ", iar i2 = " << i2 << endl;
schimba(i1, i2);
cout << "Dupa schimbare, i1 = " << i1 << ", iar i2 = " << i2 <<
endl;


char c1 = 'a', c2 = 'b';
cout << "Initial, c1 = " << c1 << ", iar c2 = " << c2 << endl;
schimba(c1, c2);
cout << "Dupa schimbare, c1 = " << c1 << ", iar c2 = " << c2 <<
endl;
return 0;
}
```

```
template <class T>
void schimba(T& var1, T& var2)
{
T temp;
temp = var1;
var1 = var2;
var2 = temp;
}
```

Definirea de parametri tip se face folosind cuvantul rezervat template urmat de <class T>, adica:

template <class T>

Aceasta constructie se mai numeste **prefix template** si spune compilatorului ca definitia sau prototipul care urmeaza este un template si ca T este un **parametru tip**. (In acest context, **class** inseamna de fapt **tip**). Definitia unei functii template este, de fapt, o multime de definitii de functii. Pentru exemplul nostru, parametrul de tip T se inlocuieste cu un nume de tip.

Cand folosim functii template, putem spune ca exprimam un algoritm general in C++.

Acesta este un exemplu simplu de **algoritmi abstracti**. Algoritmii abstracti ignora detaliile, continand partea semnificativa a algoritmului. Daca functia are mai multi parametri tip atunci declaratia template-ului se generalizeaza imediat.

Pentru doi parametri, scriem: *template <class T1, class T2>*

Fie programul C++:

// Ilustrarea unor situatii neobisnuite

```
#include <iostream.h>
template <class T>
void f(T a, T b)
{
        cout << a << b;  }

int main()
 {
  f(40000, 6);
  f(1, 2.3);
  f(1, 'a');
  f(2.3, 1);
 }
```

7.2. Template-uri pentru clase abstracte

Sintaxa pentru template-urile claselor este aceeasi ca template-urile functiilor. Ele sunt precedate de

template <class Parametr_Tip>.

Exemplul 4:

Consideram o clasa cu obiecte perechi de valori de tip T, unde T este tip abstract de date (putem pune alt cuvant in loc de T, dar asa se face in mod traditional).

```
template <class T>
 class Pereche
 {
      public:
                Pereche();
                Pereche(T val1, T val2);
                void seteaza_element(int pozitie, T valoare);
                //Preconditie: pozitie este 1 sau 2
                //Postconditie: Pozitia indicata a fost setata
     catre valoare

                T obtine_element(int pozitie) const;
                //Preconditie: pozitie este 1 sau 2
                //Returneaza valoarea pozitiei indicate

        private:
                T primul;
                T al_doilea;
 };
```

Declararea unor obiecte de tip Pereche <class T> pot fi:

Pereche <int> scor;

Pereche <char> locuri;

De exemplu setarea scorului Romania-Germania (3-0) se face astfel:

scor.seteaza_element(1,3);

scor.seteaza_element(2,0);

Functiile membru pentru o clasa template sunt definite in acelasi mod ca si functiile membru pentru clasele obisnuite. Singura diferenta este ca definitiile functiilor membre sunt ele insele template-uri.

Exemplul 5:

//definim functia membru seteaza_element si constructorul cu 2 argumente

```
template <class T>
void pereche <T>::seteaza_element (int pozitie, T valoare)
{
if (pozitie == 1)
primul = valoare;
else
if (pozitie == 2)
al_doilea=valoare;
else
{
cout << "Eroare: Pozitie inlegala in pereche\n";
exit(1);
}
}
template <class T>
Pereche <T>::Pereche(T val1, T val2)
{
primul = val1;
al_doilea = val2;
}
```

Atentie !!! Numele clasei din fata operatorului rezolutiei de domeniu este **Pereche <T>** si nu simplu **Pereche**.

Exemplul 6:

In continuare, prezentam un program C++ care descrie o clasa template a caror obiecte sunt liste. Listele sunt structuri inlantuite de orice tip (int, double, char, ...)

// Header ce descrie clasa Lista, prototipurile functiilor si

// operatorilor.

// Acest header se mai numeste si fisier de interfata.

// Obiecte de tip Lista pot fi elemente de orice tip pentru care sunt

// definiti operatorii << si =. Toate articolele unei liste trebuie sa

// fie de acelasi tip. O lista poate avea "max" elemente de tip

// Nume_Tip, dupa cum urmeaza:

// Lista <Nume_Tip> obiect(max);

```
#include <iostream.h>


template <class T>
class Lista
{
public:
Lista(int max);
// Initializeaza obiectul la lista vida care poate avea maxim "max" elemente de tip T


~Lista();
// Returneaza toata memoria dinamica folosita de obiect in heap
```

int length() const;

// Returneaza numarul de elemente ale listei

void adauga(T element_nou);

// Preconditie: Lista nu este completa

// Postconditie: "element_nou" va fi adaugat la lista

int completa() const;

// Returneaza true daca lista este completa (adica nu mai sunt elemente de adaugat)

void sterge();

// Sterge toate elementele listei astfel incat lista devine vida

friend ostream& operator <<(ostream& iesire,

const Lista <T>& lista);

// Supraincarca operatorul << care va fi folosit la afisarea listei. Articolele sunt afisate cate unul pe linie.

// Preconditie: Daca "iesire" este un flux fisier de iesire, atunci "iesire" este deja conectat la un fisier.

private:

T *element; // pointer catre un sir dinamic care memoreaza lista

```
int lungime_maxima; // numarul maxim de elemente permise de lista
int lungime_curenta; // numarul de elemente curente ale listei
};
```

Fisier de implementare care contine definitiile tuturor functiilor si operatorilor de supraincarcare.

```
#include <iostream.h>
#include <stdlib.h>
#include "probl_6.h"

// Foloseste stdlib.h
template <class T>
Lista <T> :: Lista(int max)
{
lungime_maxima = max;
lungime_curenta = 0;
element = new T[max];
if (element == NULL)
{
cout << "Eroare: Memorie insuficienta.\n";
exit(1);
}
```

```
}


template <class T>
Lista <T> :: ~Lista()
{
delete [] element;
}


template <class T>
int Lista <T> :: length() const
{
return (lungime_curenta);
}


// Foloseste iostream.h si stdlib.h
template <class T>
void Lista <T> :: adauga(T element_nou)
{
if (completa())
{
cout << "Eroare: adaugare la o lista completa.\n";
exit(1);
}
else
{
element[lungime_curenta] = element_nou;
```

```
lungime_curenta ++;
}
}


template <class T>
int Lista <T> :: completa() const
{
return (lungime_curenta == lungime_maxima);
}


template <class T>
void Lista <T> :: sterge()
{
lungime_curenta = 0;
}


// Foloseste iostream.h
template <class T>
ostream& operator <<(ostream& iesire, const Lista <T>& lista)
{
for (int i = 0; i < lista.lungime_curenta; i++)
cout << lista.element[i] << endl;
return iesire;
}
```

Program care demostreaza folosirea clasei template Lista

```
#include <iostream.h>
#include <stdlib.h>
#include "probl_7.h"


// Foloseste stdlib.h
template <class T>
Lista <T> :: Lista(int max)
{
lungime_maxima = max;
lungime_curenta = 0;
element = new T[max];
if (element == NULL)
{
cout << "Eroare: Memorie insuficienta.\n";
exit(1);
}
}


template <class T>
Lista <T> :: ~Lista()
{
```

```
delete [] element;
}


template <class T>
int Lista <T> :: length() const
{
return (lungime_curenta);
}


// Foloseste iostream.h si stdlib.h
template <class T>
void Lista <T> :: adauga(T element_nou)
{
if (completa())
{
cout << "Eroare: adaugare la o lista completa.\n";
exit(1);
}
else
{
element[lungime_curenta] = element_nou;
lungime_curenta ++;
}
}
template <class T>
int Lista <T> :: completa() const
```

```
{
return (lungime_curenta == lungime_maxima);
}


template <class T>
void Lista <T> :: sterge()
{
lungime_curenta = 0;
}


// Foloseste iostream.h
template <class T>
ostream& operator <<(ostream& iesire, const Lista <T>& lista)
{
for (int i = 0; i < lista.lungime_curenta; i++)
cout << lista.element[i] << endl;
return iesire;
}


main()
{
Lista <int> prima_lista(2);
prima_lista.adauga(7);
prima_lista.adauga(10);
cout << "prima_lista = \n" << prima_lista;
```

```
Lista <char> a_doua_lista(10);
a_doua_lista.adauga('T');
a_doua_lista.adauga('E');
a_doua_lista.adauga('M');
a_doua_lista.adauga('P');
a_doua_lista.adauga('L');
a_doua_lista.adauga('A');
a_doua_lista.adauga('T');
a_doua_lista.adauga('E');
cout << "a_doua_lista = \n" << a_doua_lista;


return 0;
}
```

```
#include<iostream.h>
template <class Ttip> Ttip compar (Ttip a, Ttip b)
{
return ( (a>b) ? a : b );
}
float compar(float a, float b);
int compar(int a, int b);
void main()
```

```
{
float a=3.14, b=7.39;
cout<<"Se compara "<<a<<" cu "<<b<<" si cel mai mare este: "<< compar(a,b) << endl;
int c=7,d=29;
cout<<"Se compara "<<c<<" cu "<<d<<" si cel mai mare este: "<< compar(c,d) << endl;
}
```

In urma executiei programului se vor afisa urmatoarele rezultate:

se compara 3.14 cu 7.39 si cel mai mare este: 7.39

se compara 7 cu 29 si cel mai mare este: 29

Definim o clasa pentru lucrul cu numerele complexe si o functie numita suma care aduna n numere complexe folosind operatorul supraincarcat “+”. Numerele complexe vor fi generate prin program.

```
#include<iostream.h>
class complex{
public:
double re;
double im;
```

```
    complex (double a=0, double b=0)
    // constructor cu initializare
    { re=a; im=b; }
    };
    complex operator+(complex c1, complex c2) //supraincarcare operator "+"
    {
    complex c;
    c.re=c1.re+c2.re;
    c.im=c1.im+c2.im;
    return c;
    }
    double& real(complex &c) {return c.re;}
    double& imag(complex &c) {return c.im;}
    template <class Comp> Comp suma(int n,Comp *p)//definire sablon penrtu suma
    //a n numere dintr-o structura comunicata prin adresa p
    {
    Comp c1;
    int i;
    for(i=0;i<n;i++) c1=c1+*(p+i);
    return c1;
    }
    void main()
    {
    complex z[4];
    int i;
    for(i=0;i<4;i++)
```

```
    {
    real(z[i])=i; imag(z[i])=-i; // formare elemente
    cout<<"numarul  z["<<i<<"]  =  "<<real(z[i])<<"  +  ("<<imag(z[i])
<<")i"<<endl;
    }
    cout<<"Suma celor 4 elemente = "<<real(suma(4,z))<<" + (";
    cout<<imag(suma(4,z))<<")i"<<endl;
    }
```

In urma executiei programului se vor afisa urmatoarele rezultate:

numarul z[0]=0+0i

numarul z[1]=1+1i

numarul z[2]=3+3i

numarul z[4]=4+4i

Suma celor 4 elemente = 6 + 6i

Problema propusa spre rezolvare

Folosind template, scrieti un program C++ care sa poata sorta siruri de intregi, caractere, double si altele. Puteti folosi metoda quicksort (cu si fara folosirea functiei predefinite qsort).

Capitolul 8 - Sistemul de intrare/ieşire folosit în C++

8.1. Noţiuni introductive

Un exemplu reusit de proiectare a unei ierarhii de clase îl constituie fluxurile. În ANSI C++ dispozitivul logic pentru citire/scriere se numeste *stream (flux)*.

Specialistii firmei Borland definesc *stream*-ul ca o clasă care înglobează orice flux de date de la o sursă la o destinatie. Atât sursa cât si destinatia pot fi oricare: consolă, zonă de memorie sau fisier. Sistemul *stream* de intrare/iesire permite înlocuirea functiilor clasice din categoria *scanf* si *printf*, cele oferite fiind mai flexibile si eficiente.

Datorită mecanismului de supraîncărcare a functiilor si operatorilor, ele permit crearea unui set propriu de metode care să manipuleze natural si să verifice în acelasi timp foarte riguros toate aspectele aferente operatiilor de intrare/iesire la nivel de obiecte.

Ierarhia de clase poartă numele **iostream** si are două clase de bază: **ios** si **streambuf.** Relatiile de mostenire ale acestora sunt:

Clasa **streambuf** are facilităti minimale ale formatelor pentru operatiile de intrare/iesire. Din această clasă sunt derivate clasele **strstreambuf** si **filebuf.**

Clasa **ios** are o complexitate mai mare:

a) oferă facilităti de tratare a erorilor apărute în operatiile de citire/scriere;

b) are un pointer spre **streambu**f si metode de interfatare cu ea;

c) dezvoltă o ierarhie de clase pentru specializarea ulterioară a operatiilor.

Pentru a întelege mai usor atributele claselor componente, vom explica prescurtările folosite în numele acestora:

- **i,** stream de intrare;

- **o**, stream de iesire;

- **str**, stream pentru zone de memorie;

- **f**, stream pentru fisier;

- **withassign**, această clasă posedă operatori suplimentari de atribuire;

- **base**, o clasă virtuală din care se vor specializa ulterior alte clase derivate.

Utilizându-se clasa **ios** drept clasă de bază virtuală sunt derivate direct clasele:

- **istream** - pentru operatii de intrare;

- **ostream** - pentru operatii de iesire.

Sunt supraîncărcati cei doi operatori de deplasare pe biti pentru manipularea datelor:

- operatorul << pentru iesire;

- operatorul >> pentru intrare.

Din cele două clase derivă clasa **iostream**, care asa cum sugerează denumirea ei, este folosită pentru operatii de intrare/iesire.

Din clasa **istream** derivă **istream_withassign**, din **ostream** derivă **ostream_withassign** si din **iostream** derivă **iostream_withassign**.

Aceste clase folosesc supradefinirea operatorului de atribuire si seturi cu functii de intrare/iesire noi.

În plus, ele contin următoarele dispozitive predefinite:

- din clasa **istream_withassign**, **cin**: dispozitiv standard de intrare consolă (asemănător lui stdin din C);

- în clasa **ostream_withassign** sunt definite dispozitivele:

- **cout**, dispozitiv standard de iesire consolă (asemănător lui **stdout** din C);

- **cerr**, dispozitiv standard de afisare a erorilor (asemănător lui **stderr** din C);

- **clog**, dispozitiv standard de afisare a erorilor dotat cu zonă tampon (fără corespondent în C).

Utilizarea acestor ierarhii de clase implică includerea fisierului ***iostream.h***.

Pentru citirea/scrierea cu format a datelor într-o zonă de memorie se folosesc clasele:

- **strstreambase**, este derivată din **ios** si în acelasi timp este clasă de bază pentru clasele:

- **ostrstream**, derivată din **ostream** si din **strstreambase**, folosită pentru conversii din format intern în siruri de caractere;

- **istrstream**, derivată din **istream** si **strstreambase** si folosită pentru extragerea dintr-un sir de caractere a variabilelor;

- **strstream**, derivată din **ostrstream** si **istrstream**; face operatii de conversie dintre sirurile de caractere si formatul intern al variabilelor si invers.

Aceste clase sunt structurate sub următoarea ierarhie:

Pentru citirea/scrierea datelor într-un fisier se folosesc clasele:

- clasa **filebuf** care este derivată din **streambuf**

- clasa **fstreambase**, derivată din **ios** si care la rândul ei este bază pentru clasele:

- **ifstream**, cuprinzând operatii de intrare pentru fisiere. În plus, ea este derivată din clasa **istream**.

- **ofstream**, cuprinzând operatii de iesire pentru fisiere. Clasa este derivată si din **ostream**.

- **fstream**, cuprinzând operatii de intrare/iesire pentru fisiere. Clasa este derivată si din **iostream**.

În Turbo C++ sunt incluse clase, care nu fac parte din standardul ANSI C++, de interfată cu consola. Acestea sunt: **constream** si **conbuf**. Rolul lor este asemănător header-ului ***conio.h***.

8.2. Operaţii cu fişiere

După cum am arătat la începutul capitolului, operatiile cu fisiere utilizează clasele:

- pentru citire, clasa **ifstream**;

- pentru scriere, clasa **ofstream**;

- pentru citire/scriere, clasa **fstream**.

Acestea sunt dublu derivate: din clasele **istream**, **ostream** si **iostream** pe de o parte si din clasa **fstreambase** (clasă specifică fisierelor, derivată la rândul ei din clasa **ios**).

Clasele dispun de mai multi constructori. La utilizarea constructorului fără parametri, se creează un stream neasociat nici unui fisier.

- Pentru deschiderea unui fisier se apelează functia membru **open**. Sintaxa este:

open (char *nume_fisier, mod_deschidere, mod_acces);

Modurile de deschidere pentru un fisier sunt:

in =0x01;

out =0x02;

ate =0x04;

app =0x08;

trunc =0x10;

nocreate=0x20;

noreplace=0x40;

binary =0x80.

- Modul **ios::in** face deschiderea pentru citire. Dacă fisierul nu există, se semnalează eroare. Este modul implicit la clasa **ifstream**.

- Modul **ios::out** deschide fisierul pentru scriere. Dacă nu există este creat, iar dacă există, vechiul continut se sterge si se începe scrierea de la începutul fisierului. Este modul implicit la clasa **ofstream**.

- Modul **ios::app (append)** permite scrierea la sfârsitul fisierului; în cazul în care fisierul nu există se semnalează eroare.

- Modul **ios::ate (at end)** permite scrierea la sfârsitul fisierului ca si **ios::app** cu deosebirea că dacă fisierul nu este creat se acceptă crearea unui fisier nou.

- Modul **ios::trunc (truncate)** dacă fisierul există, continutul lui este sters si se creează unul nou. Acest mod este implicit dacă **ios::out** este activ iar **ios::ate** si **ios::app** sunt inactivi.

- Modul **ios::nocreate** face deschiderea fără a permite crearea fisierului; dacă fisierul nu există, se înregistrează eroare.

- Modul **ios::noreplace** face deschiderea fără a permite înlocuirea continutului scris anterior. Dacă fisierul există si **ios::out** este activ iar

ios::ate si ios::app sunt inactivi se semnalează eroare; are rolul de a proteja fisierul care există pe disc.

- Modul **ios::binary** face deschiderea în mod binar; cea implicită este în mod text.

Pentru activarea mai multor parametri simultan se foloseste operatorul | **"sau binar"**.

Exemplul 1: Iată câteva variante de deschidere:

fstream f;

a) f.open("test.txt",ios::in); // Deschidere pentru citire

b) f.open("test.txt",ios::out); // Deschidere pentru scriere

c) f.open("test.txt",ios::in|ios::out); // Deschidere citire-scriere

d) f.open("c:\\image\\card.bmp",ios::in|ios::app|ios::binary| ios::noreplace); // Deschidere in mod binar,pentru citire si adaugare

e) f.open("c:\\test.txt",ios::ate); // Deschidere pentru adaugare la sfarsit

Iată câteva cazuri de deschideri incorecte:

a) f=open("c:\\test.txt",ios::ate); // Nu este folosita ca metoda

b) open("c:\\test.txt",ios::ate); // in primele trei

c) open(f,"c:\\test.txt",ios::ate); // cazuri

d) f.open("cart.txt",ios::ate|ios::out);// Se folosesc moduri incompatibile de deschidere

Pentru mod_acces sunt definite următoarele valori:

0 - fisier normal, fără restrictii de acces;

1 - fisier protejat la scriere ("read-only");

2 - fisier ascuns ("hidden");

4 - fisier sistem ("system");

8 - fisier arhivă ("archive").

Specificarea parametrilor mod_deschidere si mod_acces este optională deoarece sunt prevăzute următoarele valori implicite:

mod_deschidere = **ios::in** pentru clasa ifstream;

mod_deschidere = **ios::out** pentru clasa oftream;

mod_acces = **0**, pentru fisier normal.

Verificarea rezultatului deschiderii se face prin comparatia variabilei stream cu pointerul **NULL**. Dacă are loc egalitatea, atunci operatia de deschidere a esuat.

Pentru închiderea fisierelor se foloseste metoda:

void close();

Exemplul 2: Iată câteva cazuri de operatii de închidere

```
fstream f;
. . . . . .
a) f.close(); //Inchidere corecta
b) close(f); //Inchidere incorecta, nu se precizeaza f ca obiect
c) close() //Inchidere incorecta, lipseste numele obiectului
```

Observatii:

a) o dată cu realizarea (desfacerea) legăturii stream-fisier, metodele open si close efectuează deschiderea (închiderea) fisierelor în cauză;

b) metoda open poate fi evitată folosind un constructor cu parametri;

c) o dată închis, stream-ului i se va putea asocia un alt fisier de lucru;

d) în caz de eroare în utilizarea fisierelor, va fi setat automat indicatorul de eroare.

Exemplul 3: Sunt echivalente următoarele operatii:

a) ifstream f("test.txt");

b) ifstream f; f.open("test.txt");

8.3. Clasa istream

Am învătat despre clasa **istream** că este o clasă specializată pentru introducerea datelor.

Exemplul 1: În secventa:

```
int n;
cin >>n;
```

se citeste de la dispozitivul standard de intrare o valoare pentru variabila n de tip întreg.

O alternativă pentru citire este functia **get()**, pentru care există mai multe variante.

O posibilitate este citirea unui caracter din dispozitivul de intrare. Caracterul poate fi si spatiu.

Sunt admise variantele:

- **istream & get (char & c)**;
- **istream & get (unsigned char & c)**;
- **int get()**;

Dacă s-a detectat sfârsitul de fisier, atunci se întoarce valoarea **EOF**.

O altă posibilitate de a folosi functia **get()** este de a citi un sir de caractere:

- **istream &get (char *sir, int nr_max, char delim = '\n')**;
- **istream &get (unsigned char *sir, int nr_max, char delim = '\n')**;

Se citesc caractere până la întâlnirea caracterului separator delim sau până la un număr de nr_max-1 caractere. Caracterul delimitator este introdus în sir.

Functia **getline** are sintaxa:

istream &getline (char * sir, int nr_max, char delim = '\n');

si citeste un sir de caractere din *stream*-ul de intrare. Citirea se face până la întâlnirea delimitatorului sau până la citirea a nrmax-1 caractere. Ultimul parametru nu e obligatoriu. Implicit valoarea sa este '\n'. Metoda este asemănătoare ultimei forme a metodei get cu deosebirea că delimitatorul nu este înscris în sir, în schimb sirului i se adaugă '\0'. Este potrivită pentru citirea liniilor de fisiere text.

Citeste linie cu linie din fisierul standard de intrare. Va tipări toate spatiile, literele si cifrele găsite, celelalte caractere fiind înlocuite cu caracterul '-'.

```
#include <iostream.h>
#include <string.h>
#include <ctype.h>
void main()
{
int i;
char text[80];
```

```
cout << "Introduceti o linie \"!!\" pentru terminare\n";
for(cin.getline(text, 80); strcmp(text, "!!")!=0; cin.getline(text, 80))
{
for(i = 0; i < strlen(text); i++)
if (isspace(text[i])||isalnum(text[i]))
cout << text[i];
else cout << "-" ;
cout << '\n';
}
}
```

Programul numără câte linii, câte caractere si câte caractere tipăribile există într-un fisier. Pentru determinarea numărului de linii se va defini separat o functie.

```
#include<string.h>
#include<iostream.h>
#include<fstream.h>
#include<ctype.h>
const nr1=120;
// functia de numarare linii
int numarare_linii(char*sir);
// functia de numarare caractere
unsigned long nr_caract(char*sir,unsigned long &nr_tip);


void main()
{char sir1[30];unsigned long n;
```

```
cout<<"Numele fisierului: "; cin>>sir1;
cout<<"Numarul de linii este: "<<numarare_linii(sir1);
cout<<"\nLungimea fisierului: "<< nr_caract(sir1,n);
cout<<"\nNumarul de caractere tiparibile este: "<<n;
}


int numarare_linii(char*sir)
{int nr=0;char l[nr1];
ifstream f;


f.open(sir, ios::in); if(!f)return -1;


while(!f.eof())
{ f.getline(l, nr1); nr++; }
f.close();
return(nr);
}


unsigned long nr_caract(char*sir,unsigned long &nrt)
{unsigned long nr; char l[nr1]; ifstream f;
nr=nrt=0;
f.open(sir, ios::in, 0);
if(!f)return -1;
```

```
while(!f.eof())
{ f.getline(l, nr1);nr+=strlen(l)+2;
for(int i=0;l[i];i++)
if(isprint(l[i]))nrt++;
}
nr-=2;f.close();
return(nr);
}
```

Calcularea CRC-16 pentru un fisier

CRC este un acronim de la "**Cyclic Redundancy Check**". Când transmitem datele între două puncte vrem să stim dacă transmiterea s-a făcut corect. Pentru aceasta sunt folosite diverse variante de verificare:

- a bitului de paritate. Aceasta constă în a calcula numărul bitilor de 1 si a trimite paritatea acestui număr o dată cu datele transmise. Din păcate, numai 50% din erorile de transmisie sunt detectate (cele care modifică un număr impar de biti).

- o generalizare a acestui procedeu este uttilizarea unei secvente de biti de paritate numite CRC. Spre exemplu, la o secventă de 16 biti, sansa de a rata interceptia erorii este de numai 1/216. Se poate demonstra că un CRC care foloseste M biti va detecta orice eroare apărută într-o secventă de 1 până la M biti consecutivi.

Şirul de biti de la intrare poate fi privit ca un polinom cu coeficientii 0 sau 1. Pentru calculul CRC-ului sunt folosite mai multe polinoame:

CCITT: x16+x12+x5+1,

AUTODIN-II:
x32+x26+x23+x22+x16+x12+x11+x10+x8+x7+x5+x4+x2+x+1,

CRC-12: x12+x11+x3+x+1.

Nu vom intra în detaliile matematice care sunt necesare explicării teoriei codurilor ciclice de redundantă; ne vom mărgini la a spune că valorile polinomului se pot calcula foarte rapid folosind operatia de SAU exclusiv pe biti, deoarece polinomul are numai coeficienti de 0 sau 1.

Alte aplicatii ale codurilor ciclice sunt: verificarea fisierelor executabile împotriva alterării de către virusi, verificarea integrității datelor arhivate etc.

```
#include <fstream.h>
#include <stdlib.h>
#define DIM 16000


unsigned char buffer[DIM];
unsigned int i,crc=0,result,c;


void main(int argc, char*argv[])
{
fstream f;
if(argc!=2)
{
cerr<<"trebuie dat ca parametru program numele fisierului
sursa!";
exit(0);
}
f.open(argv[1],ios::in|ios::binary);
if(!f)
{ cerr<<argv[1]<<" nu se poate deschide";exit(0); }
```

```
do
{
f.read((char*)&buffer,DIM);
result=f.gcount();
for(c=0;c<result;c++)
{
crc^=buffer[c]<<8;
for(i=0;i<8;i++)
if(crc&0x8000) crc=(crc<<1)^4129
/* x^16+x^12+x^5+1 */;
else crc<<=1;
}
}while (result==DIM);
cout<<"\nCRC-16 : "<<crc;
}
```

Functia **read**, de citire în mod binar, citeste un sir de maxim dim_max caractere. Sintaxa ei este:

istream & read (char * sir, int dim_max);

Functia **gcount** numără câte caractere s-au citit la ultima citire. Este apelată după utilizarea functiilor **get, getline** si **read**.

Are sintaxa:

int gcount();

Pentru aflarea pozitiei curente de citire ("tell get") se poate folosi functia:

long tellg();

Pentru modificarea pozitiei curente("seek get") se folosesc functiile:

seekg(long deplasare, int reper);

seekg(long deplasare);

În primul caz, reperul folosit poate avea trei forme (**ios::beg**, **ios::cur**, **ios::end**): fată de începutul fisierului, fată de pozitia curentă si fată de sfârsitul fisierului. La folosirea cu un singur parametru se consideră reperul fată de începutul fisierului.

8.4. Clasa ostream

Clasa **ostream**, asa cum stim, este specializată în operatii de iesire.

Exemplul 1: Afisarea continutului unei variabile de tip întreg:

```
int n;
cout << " n= " << n;
```

Este afisat n.

În afară de operatorul de insertie, clasa mai contine următoarele metode:

ostream & put(char c); afisează caracterul c.

ostream & write(const char * sir, int nr_caractere)
face scrierea unui buffer într-un stream.

functia **tellp** întoarce pozitia curentă de scriere ("Put"). Are sintaxa:
long tellp( );

functia **seekp** setează o anumită pozitie pentru scriere.
ostream & seekp(long deplasare, int reper);
ostream & seekp(long deplasare);

În încheierea capitolului, vom prezenta varianta obiectuală de utilizare a consolei prin folosirea claselor **conbuf** si **constream** din header-ul *constrea.h*. Întrucât metodele clasei se folosesc asemănător header-ului *conio.h,* nu mai dăm detalii despre componentele clasei. Vom prezenta doar un scurt program de exemplificare:

Se va utiliza clasa **conbuf** din care se instantiază 3 obiecte reprezentând întregul ecran si 2 ferestre: prima cu coordonatele (1, 1, 59, 15), fond alb si cerneală albastră si a doua cu coordonatele (40, 10, 80, 25).

```
#include<iostream.h>
#include<constrea.h>
void main()
{
conbuf f, f1, f2;
f.textbackground(BLUE);
f.clrscr();


f1.window(1, 1, 59, 15); // Afisare in prima fereastra
f1.textbackground(WHITE);
f1.textcolor(BLUE);
f1.clrscr();
f1.gotoxy(20, 10);
cout<<"afisare 1"; cin.ignore(10, '\n');
// Schimbare fereastra, afisare in a doua fereastra
f2. window(40, 10, 80, 25);
f2.textattr(BLACK+RED*16);
f2.clrscr();
f2.gotoxy(10, 10);
cout<<"AFISARE 2";
cin.ignore(10, '\n');


f.window(1, 1, 80, 25); // restaurarea ecranului initial
f.clrscr();
}
```

Atragem atentia că implementarea clasei **conbuf** este fragilă: încercati instantierea clasei **conbuf** într-o functie exterioară apelată din main după

linia de comentariu... (la mine cel putin, s-a blocat de multe ori).

Exemple de utilizare

Programul face copierea a două fisiere caracter cu caracter.

```
#include<stdlib.h>
#include<iostream.h>
#include<fstream.h>
void main()
{
char c;
ifstream f1; ofstream f2;
// Verificarea pentru deschiderea celor doua fisiere
f1.open("fis1.cpp", ios::in,0);
if(!f1){cerr<<"Eroare la deschidere fisier 1"; exit(-1); }
f2.open("fis2.cpp", ios::out, 0));
if(!f2)
```

```
{cerr<<"Eroare la deschidere fisier 2"; exit(-2); }
while (f1.get(c)!=0)
f2.put(c); // Copierea
f1.close();
f2.close(); // Inchiderea celor doua fisiere
}
```

Scrie si citeste un sir de numere întregi într-un fisier binar folosind metodele **read** si **write**:

```
#include<fstream.h>
#include<iostream.h>


void main()
{
int x[6]={0, 100, 200, 300, 400, 500}, y[6];
fstream fis("test.txt", ios::in|ios::out|ios::binary );
if(!fis)
{cerr<<"Eroare la deschiderea fisierului "; return;}


fis.write((char*)&x, sizeof(x));
fis.seekp(0, ios::beg);
fis.read((char*)&y, sizeof(y));
```

```
for(int i=0; i<6; i++)
cout<<y[i]<<" ";
fis.close();
}
```

Programul concatenează n fisiere de tip text.

```
#include<fstream.h>
#include<iostream.h>


main()
{
ofstream f1;
ifstream f2;
char s1[30]="s1.txt", s2[92];
char linie[120];
int i, n=22;
cout<<"Nume fisier destinatie: "; cin >> s1;
f1. open(s1, ios::out);
cout<<"Numarul de fisiere "; cin >> n;
for(i=1; i<=n; i++)
{
cout<<"Fisier sursa "<<i<<": "; cin>>s2;
f2.open(s2, ios::in);
if(!f2)continue;
```

```
f1<<"Fisierul "<<i<<": "<<s2<<"\n\n";
while(!f2.eof())
{
f2.getline(linie, 120);
f1<<linie; f1<<'\n';
}
f2.close();
}
f1.close();
return 0;
}
```

8.5. Operaţii de formatare la intrare/ieşire

În ANSI C++ este admisă formatarea datelor pentru operatiile de intrare/iesire. Aproape toate operatiile de formatare, care erau permise cu ajutorul functiilor printf si scanf, sunt permise si în C++. De exemplu, se poate alege baza de numeratie a unui număr sau se poate stabili mărimea unui câmp sau numărul de zecimale.

Stabilirea formatului se poate face prin intermediul:

- indicatorilor de format si a functiilor aferente;
- manipulatorilor de format;
- functiilor membre din ios.

8.5.1. Facilităti pentru controlul formatului folosind indicatori

Pentru stabilirea modului de afisare a datelor se foloseste un set de indicatori pentru a controla câte unul din modurile de acces, ocupând câte o pozitie în reprezentarea binară a cuvântului de control **ios::x_flags**. Se pot controla: alinierea datelor, baza de numeratie, formatul pentru numere reale etc. Acestor indicatori, membri ai clasei **ios**, li se atribuie un nume astfel:

```
enum {
skipws=0x0001, // sare peste spatii la introducerea datelor
left=0x0002, // aliniere la stanga
right=0x0004, // aliniere la dreapta
internal=0x0008, //spatii intre semn si numar, pana la dim. max.
dec=0x0010, // conversie in baza 10
oct=0x0020, // conversie in baza 8
hex=0x0040, // conversie in baza 16
showbase=0x0080, // afiseaza explicit baza de numeratie
showpoint=0x0100, // afisare punct zecimal
uppercase=0x0200, //afiseaza majuscule pentru reprezentarea in baza 16
showpos=0x0400, // scrie + inaintea numerelor pozitive
scientific=0x0800, // scrie numerele in format exponential
fixed=0x1000, //scrie numerele reale in format cu virgula fixa
unitbuf=0x2000, // goleste tamponul dupa fiecare scriere
stdio=0x4000 //goleste tamponul dupa scrierea in stdout sau stderr
};
```

8.5.2. Functii de utilizare a indicatorilor

- Pentru activarea anumitor optiuni de format la afisare se foloseste functia **setf**. Ea are forma:

long setf(long);

si activează indicatorii de format specificati de argument. Metoda întoarce valoarea anterioară a cuvântului de control al formatului de afisare.

- Functia mai admite varianta:

long setf(long indicatori_activati,long indicatori_selectati);

Indicatorii specificati în prima listă sunt activati numai dacă se găsesc în a doua listă. Indicatorii din a doua listă care nu sunt mentionati si în prima listă sunt dezactivati.

- Functia **unsetf** are rolul de a dezactiva indicatorii specificati de argument si întoarce valoarea anterioară a cuvântului format. Are sintaxa:

long unsetf (long);

Exemplul 1: Programul va afisa reprezentarea numerelor de la 1 la o valoare maximă n în bazele 8, 10 si 16.

```
#include<iostream.h>
void main()
{
cout<<"Introduceti valoarea maxima ";
int n;
cin>>n;
char c;
cout.setf(ios::showbase|ios::right);
// Afiseaza baza si face alinierea la dreapta
for(int i=1; i<=n; i++)
{
cout.setf(ios::dec); cout<<i; //seteaza afisarea in baza 10
cout.setf(ios::oct); cout<<' '<<i; // seteaza afisarea in baza 8
cout.setf(ios::hex); cout<<' '<<i<<endl;//seteaza afisarea bazei 16
if(i%24==0)cin>>c; //intrerupe programul pana la apasarea tastei <Enter>
}
}
```

- Functia **width**, folosită pentru a afla sau a seta dimensiunea de afisare numai pentru operatia curentă (nu si pentru operatiile următoare). Ea are două forme:

int width();

int width(int dim);

În primul caz, se întoarce valoarea curentă a dimensiunii de afisare, iar în al doilea caz se modifică dimensiunea de afisare si se întoarce valoarea anterioară.

- Functia **fill** folosită pentru a afla sau a seta caracterul de umplere a câmpurilor de afisare. Ea are două forme:

char fill();

char fill(char c);

În primul caz, se întoarce valoarea curentă a caracterului de umplere, iar în al doilea caz, se modifică caracterul de umplere si se întoarce valoarea anterioară.

- Functia **precision**, folosită pentru aflarea sau setarea preciziei pentru numere reale, are două forme:

int precision();

În primul caz, se întoarce valoarea curentă a preciziei.

int precision(int nr);

În al doilea caz, se setează precizia si se întoarce valoarea sa anterioară.

Precizia, alinierea si caracterul setat pentru umplere se păstrează până la următoarea schimbare explicită, pe când dimensiunea câmpului de afisare setată cu functia width se păstrează numai pentru prima valoare afisată.

Exemplul 2: Vom încerca acum să dăm un exemplu pentru utilizarea modurilor de aliniere:

```
#include<iostream.h>
void main()
{
cout.fill('|');
cout.setf(ios::left);
cout.width(25); cout<<"Test 1 !";
cout<<'\n';

cout.fill('*');
cout.setf(ios::right);
cout.width(15); cout<<"Test 2 !";
cout<<'\n';
```

```
cout.fill('.');
cout.width(15);
cout<<"Test 3 !";
cout<<'\n';


cout.setf(ios::left);
cout.width(15);
cout<<"Test 4 !";
cout<<'\n';


cout.precision(4);
cout.width(10);
cout<<-1/7.<<"\n";
cout.flags(ios::internal);
cout.width(10);
cout<<-1/7.<<"\n";


cout.width(10);
cout<<-2/7.<<"\n";
}
```

Rezultatul executiei programului va fi:

Test 1 !||||||||||||||||

*********Test 2 !**

.....Test 3 !

Test 4 !.....

-0.1429...

-...0.1429

-...0.2857

Aparitia punctelor după semnul minus de pe ultimele două linii se datorează functiei de setare a caracterului de umplere si indicatorului internal.

- Functia **flags** este folosită atât pentru activarea indicatorilor cât si pentru aflarea stării lor. Are două forme. În prima formă se foloseste astfel:

long flags();

si are rolul de a întoarce caracteristicile formatului curent.

A doua formă are sintaxa:

long flags(long indicatori);

si are rolul de a seta indicatorii. Valoarea întoarsă de functie este valoarea anterioară a cuvântului de control.

Apelul flags(0) resetează formatul la valoarea implicită.

Exemplul 3: Vom da un scurt exemplu de utilizare a functiei flags pentru acelasi caz de afisare a numerelor în bazele 8, 10 si 16.

```
#include<iostream.h>
void main()
{
cout.width(10);
for(int i=12; i<=17; i++)
{
cout.flags(ios::oct);
cout.width(10);
cout<<i<<" ";
cout.flags(ios::dec);
cout.width(10);
cout<<i<<" ";
cout.flags(ios::hex|ios::showbase|ios::uppercase);
cout.width(10);cout<<i<<"\n";
}
}
```

14 12 0XC

15 13 0XD

16 14 0XE

17 15 0XF

20 16 0X10

21 17 0X11

Vă propunem ca exercitiu scrierea unei functii care să verifice la un moment dat starea indicatorilor activati.

8.5.3. Controlul formatului folosind manipulatori

Manipulatorii reprezintă o nouă cale de modificare a parametrilor de format.

Ei sunt definiti în header-ul *iomanip.h* si sunt necesari în operatiile pentru **stream**-urile de intrare/iesire.

Pe lângă utilizarea predefinită, ei oferă în plus prin redefinirea lor posibilitatea sintetizării unei secvente de mai multe operatii de I/O, simplificând astfel scrierea codului si protejând programul de eventualele erori.

În functie de utilizarea parametrilor, ei pot fi de două tipuri: cu parametri si fără parametri.

a) Manipulatorii fără parametri pot avea una din formele:

dec - activează bitul dec (pentru operatii de intrare/iesire);

hex - activează bitul hex (pentru operatii de intrare/iesire);

oct - activează bitul oct (pentru operatii de intrare/iesire);

ws - activează bitul skipws (pentru intrare);

endl - inserează caracterul '\n' pentru sfârsitul liniei;

ends - inserează caracterul '\0' pentru sfârsit de sir;

flush - goleste tamponul de iesire.

b) Manipulatorii cu parametri sunt:

setprecision(int dim) - stabileste precizia pentru afisare;

setfill(caracter) - stabileste caracterul de umplere;

setw(int) - stabileste dimensiunea câmpului de format numai pentru următoarea operatie;

setbase(baza) - stabileste baza de numeratie care poate fi 8, 10 sau 16;

setiosflags(val) - setează cu 1 toti bitii indicatori specificati în argument;

resetiosflags(val) - setează toti bitii indicatori la valoarea 0.

Exemplul 1: Dăm o altă variantă de reprezentare a numerelor în bazele 8, 10 si 16 afisând un tabel cu numere cuprinse între 1 si n.

```
#include<iostream.h>
#include<iomanip.h>
void main()
{
cout<<"Introduceti valoarea maxima ";
int n;
```

```
cin>>n;
cout<<setiosflags(ios::showbase|ios::left);


for(int i=1; i<=n; i++)
{
cout<<setbase(10)<<setw(15)<<i;
cout<<setbase(8);
cout<<setw(15)<<i;
cout<<setbase(16);
cout<<setw(15)<<i<<endl;
}
}
```

Exemplul 2: Stabilirea preciziei cu ajutorul manipulatorilor:

```
#include<iostream.h>
#include<iomanip.h>
#define afisare cout<<setw(15)<<x<<endl;
void main()
{float x=12.34567;
afisare
cout<<setprecision(3);
afisare
cout<<resetiosflags(0);
cout<<setprecision(3);
cout<<setiosflags(ios::scientific|ios::left);
```

afisare

Pe ecran se va afisa:

12.34567

12.346

1.234e01

Exemplul 3: Adăugarea de caractere introduse de la tastatură la sfârsitul unui fisier până la introducerea caracterului Ctrl-Z.

```
#include<iostream.h>
#include<fstream.h>
main(int argc,char **argv)
{
fstream f(argv[1],ios::app);

if(argc!=2)
{
cerr<<"Introduceti un nume de fisier";
}
if(!f)
{
cerr<<"Eroare la deschiderea fisierului!"; return -1;
}
```

```
    cin.unsetf(ios::skipws); /* anuleaza optiunea de salt peste caractere albe*/
    char c;
    while(cin>>c) f<<c;
    f.close();
    return 0;
    }
```

8.5.4. Redefinirea operatorilor de intrare/iesire si a manipulatorilor

Atât manipulatorii cât si operatorii de intrare/iesire pot fi redefiniti în scopul utilizării eficiente a operatiilor de intrare/iesire.

Până acum pentru introducerile de la tastatură foloseam operatorul >>. Acum este cazul să subliniem că operatorul este membru component al clasei **istream**, având un rol special în extragerea datelor (poate fi folosit atât pentru tipuri de date predefinite ale limbajului cât si pentru tipuri de date ale utilizatorului) si este supradefinit în forma următoare:

istream & operator >>(tip_baza&);

El se mai numeste si operator de extragere dintr-un *stream* (sau extractor).

În privinta utilizării operatorului de afisare este cazul să deducem că utilizarea sa este aceeasi cu cea a lui **cout**. În primul rând să amintim că se redefineste operatorul << (deplasare la stânga) pentru operatiile de iesire standard. Pot fi folosite pentru afisare atât expresii de un tip predefinit cât si expresii de un tip definit de utilizator. Operatorul << este redefinit sub forma:

ostream & operator << (tip_baza&)

si se mai numeste si operator de insertie (sau insertor).

Atragem atentia că tipurile definite de utilizator folosite la operatorii de insertie si extragere trebuie să fie clase si să redefinească explicit operatorul.

Exemplul 1: Supradefinirea insertorului si extractorului pentru introducerea si afisarea matricilor. Secventa de redefinire este următoarea:

```
#include<iostream.h>

class matrix
{ public:
int mat[30][30], m, n;

friend ostream & operator<<(ostream&, matrix);
friend istream & operator>>(istream&, matrix&);
//...

};

//...

istream & operator>>(istream&f, matrix &a)
{f>>a.m>>a.n;
```

```
for( int i=1; i<=a.m; i++)
for( int j=1; j<=a.n; j++)
f>>a.mat[i][j];
return f;
}

ostream & operator<<(ostream&f, matrix a)
{for(int i=1; i<=a.m; i++)
{for(int j=1; j<=a.n; j++)
f<<a.mat[i][j];
f<<'\n';
}
return f;
}
```

Utilizarea pentru citire si scriere poate fi:

```
matrix a1, a2, a3;
cin>>a1; cin>>a2;
. . . . .
cout<<a3;
```

Observăm că functiile operator << si >> se încheie cu o instructiune return f. Aceasta are rolul de a permite operatii de intrare/iesire multiple în cadrul unei singure instructiuni: cin>>a1>>a2>>a3 sau cout<<a1<<a2<<a3.

Manipulatorii sunt facilități de formatare, adaptate caracteristicilor C++, care tratează operatiile de intrare/iesire specifice obiectelor. Pentru a face o tratare mai bună, ei admit redefinirea.

Vom încerca să studiem acest mod.

Exemplul 2: În următorul program vom încerca să exemplificăm definirea manipulatorilor fără parametri. Vom defini un manipulator pentru afisarea de fiecare dată a câte unui mesaj la introducerea si afisarea unei optiuni.

```
#include<iostream.h>
char opt;


istream &prompt(istream &in)
{cout<<"Introduceti optiunea : ";
in>>opt;
return in;
}


ostream &prompt(ostream&out)
{ out<<"Optiunea este: "<<opt;
return out;
}
```

```
void main()
{ cin >>prompt;
cout<<prompt;
}
```

Manipulatorul din acest exemplu se numeste prompt. El este o functie supradefinită pentru operatiile de intrare si iesire.

Pentru definirea unui manipulator cu parametri, problema este mai dificilă. Aceasta deoarece compilatoarele nu au o variantă unitară de tratare. Pentru aceasta recomandăm consultarea documentatiei firmei care detine licenta compilatorului de C++.

8.6. Tratarea erorilor

Pentru orice obiect de tip **stream** există un cuvânt de stare care descrie modul în care a decurs operatia precedentă. În cuvântul de stare există 4 biti indicatori. Pentru a accesa indicatorii este definit tipul enumerare er_state:

```
public: enum er_state
{
goodbit=0,
eofbit=1,
failbit=2,
badbit=4,
hardfail=8
};
```

Valorile folosite au următoarele semnificatii:

goodbit - operatie corectă. În acest caz nu este activat nici un bit indicator;

eofbit - indicator pentru sfârsit de fisier;

failbit - operatie esuată;

badbit - operatie invalidă;

hardfail - eroare care nu poate fi recuperată.

Pentru citirea si testarea cuvântului de stare avem functiile:

good() - întoarce o valoare nenulă dacă operatia s-a încheiat fără erori;

eof() - verifică sfârsitul unui fisier (bitul eofbit este setat);

fail() - arată dacă operatia efectuată a esuat (bitul failbit este setat);
rdstate() - întoarce valoarea cuvântului de stare.

Pentru setarea indicatorului de stare al unui stream avem functia:
clear(int i=0) - la valoarea implicită 0 se sterg toti indicatorii.

Exemplu: Verificarea aparitiei erorilor de intrare/iesire într-un program.

```
#include<iostream.h>
#include<fstream.h>

void verif_stare();
ifstream in;

main(int argc, char*argv[])
{ if(argc!=2)
{cout<<"\n Utilizare: AFISEAZA <nume_fisier> \n"; return -1; }
in.open(argv[1]);
if(!in) {cerr<<"Nu se poate deschide fisierul de intrare";
return -1;
}
char c;
// Se incearca executia unei operatii pana la aparitia unei erori
while(in.good() && !in.eof() && in.get(c))
```

```
cout<<c;
verif_stare();
return 0;
}
void verif_stare()
{int i;


i=in.rdstate();
if(i&&ios::eofbit) cerr<<"\nS-a ajuns la sfarsit de fisier !\n ";
else
if(i&&ios::failbit) cerr<<"\nEroare nefatala ! \n";
else
if(i&&ios::badbit) cerr<<"\nEroare fatala ! \n";
}
```

8.7. Exemplu recapitulativ

Să se implementeze o clasă cu operatiile necesare sirurilor de caractere.

Pentru a face cât mai usoară utilizarea sirurilor de caractere în C++ vom folosi redefinirea operatorilor.

Mai întâi sirul de caractere va avea dimensiune variabilă iar clasa sir de caractere (numită string) va avea trei constructori:

- un constructor de copiere

- un constructor de stocare a unui sir de ccaractere dat ca parametru

- un constructor oarecare în care se face alocarea sirului după dimensiunea dată ca parametru.

În lipsa specificării parametrului de dimensiune se va considera implicit dimensiunea 80.

Operatorii definiti vor fi:

+, concatenarea a două siruri, operatorul este de tip **friend**;

+=, concatenarea sirului initial cu un alt sir;

=, atribuirea realizată atât pentru obiecte de tip sir cât si pentru tablourile de caractere;

[ ], căutarea unui sir de caractere într-un sir;

( ), extragerea unui sir de caractere între 2 pozitii date.

<<, operatorul de afisare a unui sir

>>, operatorul de citire a unui sir.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>


class string
{ char*s;
public:
string(int=80);
string(char*);
```

```
string(string &);
friend string operator+(string,string);
string operator=(char *);
string operator=(string);
string operator+=(string);
string operator[](char*);
string operator()(int, int);
friend istream &operator>>(istream &f,string&);
friend ostream &operator<<(ostream &f,string);
~string();
};


string::string(int dim)
{
s=new char[dim];
}


string::string(char*sir)
{
s=strdup(sir);
}


string::string(string &str)
{
s=strdup(str.s);
}
```

```
string::~string()
{delete s;}


string operator+(string s1,string s2)
{char*p;
p=new char[strlen(s1.s)+strlen(s2.s)+1];
strcpy(p,s1.s);strcat(p,s2.s);
string s(p);
return s;
}


string string::operator=(string sir)
{
s=strdup(sir.s);
return *this;
}


string string::operator=(char *str)
{ return string(s=strdup(str));}


string string::operator+=(string sir)
{
strcat(s,strdup(sir.s));
```

```
return *this;
}

string string::operator[](char*sir)
{
s=(strstr(s,sir));
return *this;
}

string string::operator()(int a,int b)
{char *c=new char[b-a+1];
int i;
for(i=a;i<=b&&i<=strlen(s);i++)
c[i-a]=s[i];
c[i-a]='\0';
s=new char [i-a];s=strdup(c);
return *this;
}

istream & operator>>(istream &f,string&str)
{
cout<<"Introduceti sirul :"; f>>str.s;
return f;
}
ostream & operator<<(ostream &f,string str)
{
```

```
f<<str.s;
return f;
}


void main()
{
clrscr();
char sir[10];
string s1(" Testare"),
s2=string(" Turbo C++");
string s=s1;
cout<<s<<"\n";
cout<<s2<<"\n";
cout<<s(4,7)<<"\n";
cout<<(s=s1+s2)<<"\n";
cout<<(s1+=s2)<<"\n";
cin>>sir;//"Turbo";
cout<<s1[sir];
}
```

Capitolul 9 - Tratarea exceptiilor

9.1. Generalităti despre tratarea exceptiilor

Mecanismul de tratare a exceptiilor constituie o solutie eficientă de detectie si corectare a erorilor generate pe parcursul executiei programelor, impusă de Stroustrup încă din implementările din 1991 si care a devenit apoi un criteriu important în evaluarea limbajelor OOP. Mecanismele au fost adoptate pe scară largă o dată cu impunerea standardului ANSI C++.

În ANSI C există următoarele posibilităti de tratare a erorilor:

a) se întoarce un cod de eroare si se continuă executia programului;

b) se afisează un mesaj de eroare si se opreste executia programului;

c) este setată o variabilă globală;

d) se apelează functii speciale destinate tratării erorilor.

Aceste moduri de tratare s-au dovedit a fi deficitare:

- fie opreau prea brusc executia programului fără a se încerca efectiv o posibilitate de recuperare a erorii;

- fie se ignora eroarea si de aici se genera în lant o secventă de erori.

Acum mecanismul de tratare a erorilor s-a dovedit a fi o automatizare a codului de remediere a erorilor. Coordonarea tuturor acestor actiuni este făcută de compilator. Aceste facilităti sunt independente de sistemul de operare ales. Bjarne Stroustrup a rezumat succint esenta mecanismului de tratare a exceptiilor astfel:

"Sistemele tolerante la erori sunt structurate pe mai multe nivele. Fiecare dintre aceste nivele tratează cât mai multe exceptii cu putintă, cu unica restrictie de a nu compromite claritatea abordării prin

supraîncărcare. Toate exceptiile netratate vor fi transmise nivelelor ierarhice superioare".

Mecanismul de tratare a exceptiilor măreste puterea de control între module de program dezvoltate independent. Structurarea tratării situatiilor exceptiilor pe mai multe nivele facilitează acumularea gradată a informatiilor necesare tratării erorii apărute. Evenimentul initial creat în momentul declansării exceptiei va fi completat la fiecare pas intermediar cu informatii specifice astfel încât, în final să se poată ajunge la determinarea completă a informatiilor necesare tratării erorii.

9.2. Moduri de utilizare a exceptiilor

Mecanismul de tratare a exceptiilor în C++ se bazează pe utilizarea următoarelor trei componente:

- try - este folosită pentru a detecta o exceptie
- catch - folosită pentru a trata eroarea
- throw - folosită pentru a o activa.

Sintaxa de utilizare este:

```
try{
// secventa de cautare a erorii
}
catch(tip_1 arg){
// prima forma de tratare a erorii
}
catch(tip_2 arg){
// a doua forma de tratare a erorii
}
```

.

.

.

catch(tip_n arg){

// forma numarul n de tratare a erorii

}

throw exceptie

Prin activare se întelege initierea unui proces prin care apelantul metodei va fi înstiintat de producerea unui eveniment neobisnuit.

Declansarea respectivei actiuni se face prin utilizarea cuvântului cheie **throw**. Acesta va avea un parametru care va furniza informatii suplimentare referitoare la tipul exceptiei. El poate fi orice tip de date, inclusiv al unei clase definite de programator.

Printr-un mecanism transparent utilizatorului, compilatorul va căuta blocul try în care este inclusă direct sau indirect (adică prin apeluri succesive de functii) metoda care a provocat exceptia. Urmează identificarea blocului catch corespunzător tipului de date transmis ca parametru lui **throw**. Apoi controlul va fi cedat respectivei portiuni de cod, care va interpreta datele descriind exceptia survenită. În cazul în care compilatorul nu va găsi nici un bloc catch care tratează exceptii de tipul activat, executia programului va lua sfârsit.

Se impun câteva observatii:

- Modul de functionare al structurilor try-catch poate fi comparat cu executarea unei instructiuni switch-case: evaluarea expresiei din **switch** este echivalentă cu executarea blocului de instructiuni **try**, iar ramurile case pot fi substituite cu blocurile catch.

În momentul întâlnirii unui bloc **try**, instructiunile vor fi executate succesiv până la încheierea lui sau până la generarea unei exceptii.

În primul caz toate blocurile de instructiuni **catch** vor fi ignorate, executia programului continuând cu prima instructiune de după secventa

try-catch.

Dacă însă executia unei instructiuni din blocul **try** generează o exceptie, compilatorul va căuta prima clauză catch capabilă să trateze o exceptie de tipul celei declansate. Se va continua cu executia respectivului bloc de instructiuni, toate celelalte blocuri de instructiuni **catch** fiind ignorate.

O problemă deosebită o reprezintă revenirea la instructiunea care a provocat exceptia. În forma propusă initial de Stroustrup, tratarea unei exceptii permitea doar căutarea automată a blocului catch corespunzător, nu si revenirea ulterioară la instructiunea care a provocat-o! Orice functie poate activa exceptii de mai multe "tipuri" (adică returnând diverse tipuri de informatii).

Un bloc **try** poate fi urmat de mai multe blocuri catch (fiecare tratând un alt tip de exceptie). O exceptie activată prin **throw** va fi tratată la nivelul celui mai recent întâlnit bloc **try**.

Buna functionare a întregului mecanism de tratare a exceptiilor depinde în mod evident de informatiile transmise de către declansatorul exceptiei către nivelul la care ea va fi tratată. Aceste informatii sunt reprezentate prin instante ale unor clase-exceptie; respectivele instante se numesc obiecte-exceptie. Definirea obictelor-exceptie se face în cadrul instructiunii **throw**.

O copie a blocului exceptie va fi transmisă blocului definit în catch într-o manieră asemănătoare modului în care obiectele sunt transmise ca parametri sau ca valori returnate. Respectiva copie va fi distrusă doar la părăsirea blocului **catch**. Pentru o corectă initializare a copiilor în cauză, nu se recomandă utilizarea constructorului implicit de copiere al clasei exceptie. Este de preferat implementarea unei versiuni proprii a respectivei metode.

Când programul detectează o situatie anormală care nu a fost tratată, se poate transfera controlul într-o altă parte a programului care a fost definită pentru a rezolva această eroare.

Aceasta este implementată prin redefinirea celor trei cuvinte cheie: **try, catch si throw**.

Blocul specificat prin **try** poate fi folosit imediat de handler-ul specificat prin **catch**. Dacă o exceptie este găsită într-un bloc **try**, controlul programului este transferat pentru cel mai recent handler executat pentru

exceptie. Programul poate astepta captarea oricărei exceptii de acest tip care este generată din orice functie. Eroarea care nu poate fi tratată în acest caz generează terminarea anormală a programului.

Blocul de cod tratat va fi prefixat de cuvântul cheie **try** si va fi cuprins între acolade. Aceasta indică folosirea în program a testării existentei exceptiilor. Dacă în timpul tratării apare încă o eroare, programul este întrerupt.

Secventele care urmează sunt, în ordine, următoarele:

- Programul caută un handler potrivit pentru exceptie. Dacă handler-ul este găsit, pe stivă se va salva adresa de declansare a erorii si contextul initial iar controlul programului este transferat handler-ului potrivit. Dacă nu a fost găsit nici un handler, atunci programul apelează functia terminate.

- Când este apelată o exceptie, atunci este initializat un obiect temporar. Alte copii pot fi generate la cererea compilatorului. De aceea este utilă definirea unui constructor de copiere pentru obiectul tratat ca tip exceptie.

Exemplul 1: Iată o scurtă secventă în care se arată cum trebuie folosită definirea unei exceptii:

```
#include<iostream.h>
void main()
{
cout<<"Testare tratari exceptii! \n ";
try{
cout<<" In interiorul blocului de tratare";
throw (1);
cout<<"mesaj dupa apel tratare";
}
catch(int i)
```

```
{
cout<<"\nAm detectat un mesaj de tratare a exceptiilor de cod:"
<<i<<endl;
}
cout<<" Sfarsit test!";
}
```

Rezultatul programului va fi:

Testare tratari exceptii!

In interiorul blocului de tratare

Am detectat un mesaj de tratare a exceptiilor cu codul 1

Observatie: Ne punem întrebarea dacă se face conversie în cazul parametrului folosit pentru tratarea exceptiei. Dacă pentru definirea lui catch am fi folosit tipul float în locul tipului int, atunci nu s-ar fi făcut apelul exceptiei.

Exemplul specifică definirea unui handler al obiectului T. Dacă argumentul este de tipul T, T&, const T sau const T&, handler-ul va accepta un obiect X pentru oricare din cazurile:

- T si X sunt de acelasi tip;

- T este un pointer si X este un pointer care poate fi convertit la T printr-o conversie standard;

- T este dintr-o clasă de bază accesibilă pentru X.

Instructiunea **goto** poate fi folosită pentru a face un transfer al controlului programului, dar nu va putea fi folosită pentru a intra în handler-

ul de întreruperi. După executia handler-ului, programul este reluat din punctul în care exceptia a fost apelată. Nici un alt handler nu va fi apelat pentru exceptia curentă.

Instructiunea **catch** poate trata orice exceptie. Dacă este utilizată, instructiunea este ultimul handler pentru blocul care va fi specificat. Limbajul C++ face posibilă specificarea oricărei exceptii pe care o functie o apelează. Specificarea exceptiei impune un sufix la declararea functiilor.

Un pointer la functie nu este afectat de exceptia functiei. Pointerul la o functie verifică doar tipul argumentelor si valoarea întoarsă dar nu apelează întreruperea specifică functiei initiale.

Exemplul 2: Exemplu pentru tratarea împărtirii la 0:

```
#include<iostream.h>

void impart(float,float);

void main()
{
float x,y;
while(1)
{
cout<<"x=";cin>>x;
cout<<"y=";cin>>y;
impart(x,y);
}
}
```

```
void impart(float a,float b)
{
try{
if(b) cout<<"\n Rezultatul este: "<<a/b;
else throw b;
}
catch b
{ cout<<"Nu este definita impartirea la 0"; }
}
```

Exemplul 3: Prezentăm în continuare următoarele cazuri de apel:

```
void f2(void) throw(); // nu va genera o exceptie

void f3(void) throw(Obiect); // Va fi apelat pentru obiecte throw

void (* fptr)(); // Pointer la functie void

fptr = f2;
```

```
fptr = f3;
```

Cazul extrem este cel de suprascriere a functiilor virtuale. Iarăsi, deoarece exceptia specificată nu este considerată o parte a acestui tip, este posibilă încălcarea regulii de proiectare a claselor.

Dacă o exceptie este apelată fără a fi specificată în lista de exceptii de până atunci, va fi apelată functia unexpected.

Exemplul 4: Exemplul următor va urmări implementarea unor exceptii care sunt folosite de către o clasă de tip listă. Se vor defini erorile:

1 - nu se poate face alocarea memoriei;

2 - element inexistent;

3 - lista plină;

4 - dimensiunea listei este necorespunzătoare.

```
#include<iostream.h>
#include<stdlib.h>


class lista
{
int *p,nel,el;
public:
lista(int n);
~lista();
void add(int);
int operator[](int n);
};
```

```
lista:: lista(int n)
{ // se verifica daca se va folosi o dimensiune incorecta
if(n<0||n>10000)
throw(4);
p=new int[n];
if(!p)throw(int(1));
nel=n;
el=0;
}


lista:: ~lista()
{ delete p; }


int lista:: operator[](int n)
{
if(nel<el || n<0) throw(int(2));
return p[n];
}


void lista:: add(int nr)
{
if(el>=nr)
throw(int(3));
else p[el++]=nr;
```

```
}



void f1()
{
lista a(10);
int n;
cin>>n;
lista b(n);
for(int i=0;i<=n-10;i++)
a. add(i);
cout<<a[0];cout<<a[-1];
}

// Alocarea unei liste cu un numar prea mare de elemente


void f2()
{ lista a(20000);}


void main()
{
cout<<" Test tratare exceptii ";
try
{
```

```
f1(); f2();
}
catch(int i)
{
switch(i)
{
case 1: cerr<<"Alocare imposibila !"; exit(-1);
case 2: cout<<"Elementul nu exista !"; break;
case 3: cout<<"Lista este pina !"; break;
case 4: cerr<<"Dimensiunea listei incorecta !";
exit(-1);break;
default: break;
}
}
}
```

Există situatii în care cunoasterea tipului obiectului-exceptie oferă suficiente informatii pentru tratarea exceptiei. În astfel de cazuri, nu mai e necesară specificarea numelui obiectului în cauză.

Dacă simpla tipărire a unui mesaj de eroare este considerată suficientă, putem să nu mai atasăm nici un nume obiectului-exceptie. Astfel de nume sunt necesare doar dacă se doreste extragerea anumitor informatii suplimentare din respectivul obiect.

Observatii:

1) Mecanismul de tratare a exceptiilor a fost creat în primul rând pentru a oferi posibilitatea unei mai bune interpretări a erorilor de executie. El

poate fi însă folosit si în alte scopuri, cum ar fi părăsirea unor bucle infinite sau a unor functii necorespunzătoare.

2) Pentru ca utilizarea mecanismului de tratare a exceptiilor să fie cu adevărat eficientă se impun două conditii:

- numărul de blocuri catch trebuie să fie sensibil mai redus decât cel al functiilor si metodelor definite în program;

- numărul exceptiilor active trebuie să fie sensibil mai redus decât cel al apelurilor de functii efectuate în timpul executiei programului; în caz contrar, aceasta indică o deficientă în tratarea erorilor.

În acest moment avem suficiente informatii pentru a putea întelege în totalitate definitia dată de Stroustrup exceptiilor:

"Tratarea exceptiilor oferă o modalitate de transferare a controlului si informatiilor necesare unui apelant nespecificat care si-a manifestat dorinta de a trata exceptiile de un anumit tip".

9.3. Clase si obiecte de tip exceptie

Pentru a întelege mai usor mecanismul de tratare a exceptiilor, vom da câteva exemple de utilizare:

- obiectele partial construite, în cazul declansării unei exceptii vor fi apelati doar destructorii corespunzând componentelor integral construite. Spre exemplu, să presupunem că avem un tablou de n elemente si că în timpul apelului constructorului elementului i survine o situatie exceptională. În acest caz se vor apela doar destructorii elementelor 0,1,. . ., i-1;

- imposibilitatea alocării dinamice de memorie, în astfel de cazuri se recomandă evitarea unor instructiuni de genul: **throw (new exceptie);** deoarece lipsa de memorie va face imposibilă crearea obiectului exceptie.

Utilizarea unei asemenea exceptii e cu totul ineficientă si riscă să aibă efecte catastrofale.

Obiectele-exceptii pot fi de trei tipuri:

a) instante ale unor clase sau tipuri de date predefinite;

b) referinte la instante ale unor clase sau tipuri de date predefinite;

c) pointeri la instante ale unor clase sau tipuri de date predefinite.

9.4. Functii predefinite pentru tratarea erorilor

Pentru cazul în care mecanismul de tratare a erorilor nu mai poate face fată situatiilor existente au fost introduse două functii predefinite:

- ori de câte ori o functie declansează o exceptie nedeclarată în cadrul blocului throw, compilatorul va apela functia unexpected. O astfel de tratare este întotdeauna binevenită deoarece declansarea unei exceptii neprevăzute este de obicei dovada unei grave erori de proiectare!

- functia terminate este apelată de către compilator în trei situatii:

a. când mecanismul de tratare a erorilor descoperă că stiva a fost alterată (coruptă) de către programul în curs de executie;

b. când mecanismul de tratare a erorilor se află în imposibilitatea identificării unui bloc catch capabil să trateze o exceptie deja declansată;

c. când o instructiune dintr-un bloc try declansează o exceptie si un destructor al unui obiect din cadrul aceluiasi bloc declansează la rândul lui încă o exceptie.

Implicit, functia unexpected apelează pe terminate care la rândul ei apelează abort. Utilizatorul îsi poate defini propriile variante pentru terminate si unexpected. Tot acest mecanism este foarte asemănător definirii unei functii utilizator de tratare a eseculului alocării dinamice de memorie (prin set_new_handler).

9.5. Aspecte particulare în tratarea exceptiilor Turbo C++

În Turbo C++ (de la versiunea 4) a fost implementată tratarea exceptiilor. De asemenea, s-a extins după modelele realizate în C++ si pentru sursele scrise în variantă C, tratarea exceptiilor.

Cuvintele cheie folosite în sintaxa în Turbo C++ sunt:

__except

__finally

__try

Atragem atentia că folosirea acestor cuvinte este valabilă doar în versiunile C. Astfel, pentru detectie, implementarea are sintaxa următoare:

__try pentru modulele în C

try pentru modulele în C++.

In Turbo C, pentru terminare este folosită sintaxa **__finally**. Cuvântul cheie **try** este urmat de un bloc de instructiuni ce reprezintă blocul de cod

asociat si indică faptul că aici este setată zona de receptionare a unei exceptii.

Dacă exceptia este generată într-un modul C, este posibil ca handler-ul să fie structurat atât ca un modul C cât si ca un modul C++.

Dacă va fi găsit un handler, atunci vor fi făcute următoarele actiuni:

- Se execută actiunea specificată de handler,

- Se ignoră generarea exceptiei;

- Se continuă căutarea pentru alte handlere;

- Dacă nu este găsit nici un handler potrivit, se apelează functia terminate;

- Dacă nu a fost apelată nici o exceptie, programul se execută normal.

Limbajul C++ defineste un standard pentru tratarea exceptiilor. Standardul asigură că puterea de proiectare a OOP este valabilă mai mult ca oricând în programele scrise în C++. Caracteristica esentială care permite tratarea acestora este posibilitatea utilizării functiilor virtuale si posibilitatea utilizării claselor pentru a defini tratarea unei exceptii.

Versiunile noi de Turbo C++ suportă specificatiile ANSI C++ pentru exceptii. Când este detectată o situatie anormală la executia programului, chiar dacă acesta ar trebui terminat, este permisă în acelasi timp interceptarea unei exceptii, diagnosticul asupra erorii si obtinerea informatiilor despre modul în care trebuie intervenit în acest punct pentru rezolvare.

De asemenea, se poate specifica ce actiuni sunt executate înainte de terminarea programului. Evenimentele generate din afara executiei programului (spre exemplu apăsarea Control-C) nu sunt considerate exceptii.

Capitolul 10 - Exemple de clase implementate în limbajul C++

Consideram definitia clasei **natural**, care permite adunarea unor intregi fara semn de orice lungime in baza 10.

```
#include<iostream.h>
#include<string.h>
#include<assert.h>

#define MAX(a,b) ((a>b) ? a:b)
#define MIN(a,b) ((a<b) ? a:b)

#define TRUE 1
#define FALSE 0
```

```
class natural{
protected:
int n;
char* data;
public:
natural() { n = 0 ; data = NULL; }
natural(char*); // constructor de initializare
natural(natural&); // constructor de copiere
natural& operator=(natural&);
~natural() {if (data) delete[] data;}
int length() const { return n; }
void resize(int);
char& operator[] (int i) { return data[i]; }
friend natural operator+(natural&, natural&);
friend natural operator-(natural&, natural&);
friend int operator<(natural&, natural&);
};


/*
```

Fie clasa derivata intreg, care permite adunarea numerelor intregi cu semn de orice lungime:

natural <- intreg

Clasa intreg are un membru suplimentar - semn - un intreg cu valoarea 0, daca numarul este pozitiv si 1, daca este negativ.

Dati definitia clasei intreg si implementati cele doua clase. Implementati in clasa intreg si operatorul << supraincarcat.

***/**

/*

Rezolvare:

Cifrele unui numar sunt retinute intr-un vector incepand cu cea mai putin semnificativa (data[0] este cea mai putin semnificativa cifra). A fost aleasa aceasta reprezentare deoarece este potrivita pentru operatiile de adunare si scadere.

Functia operator+() simuleaza adunarea manuala, calculand pentru fiecare rang suma si transportul in rangul urmator. Se porneste de la cea mai putin semnificativa cifra si se continua pana cand transportul este zero si am ajuns la ultima cifra din ambele numere.

Pentru a aduna numere intregi a fost definita functia natura::operator-() care scade doua numere cu conditia ca rezultatul sa fie numar natural. Adunarea a doua numere intregi revine la a face suma/diferenta modulelor acestor numere.

***/**

```
natural::natural(char *sir)
{
n=strlen(sir);
data = new char[n];
```

```
for(int i=0; i<n ;i++)
data[i] = sir[i-i-1] - '0';
}


natural::natural(natural& x)
{
n=0;
data=NULL;
resize(x.length());
memcpy(data,x.data,n);
}


natural& natural::operator=(natural& x)
{
if(this != x)
{
resize(x.lenght());
memcpy(data,x.data,n);
}
}


void natural::resize(int new_size)
{
char *new_data = new char[new_size];
memcpy(new_data, data, MIN(new_size,n));
if(data) delete[] data;
```

```
data = new_data;
n = new_size;
}


natural operator+(natural& a, natural& b)
{
natural c;
c.resize(MAX(a.length(),b.lenght()) + 1);
int transport = 0;
int digit, i;
for(c.n=0; c.n<a.length() || c<b.length() || transport >0; c.n++)
{
digit = transport;
if(c.n < a.length()) digit+=a[c.n];
if(c.n < b.length()) digit+=b[c.n];
c[c.n]=digit%10;
transport=digit/10;
}
c.resize(c.n);
return c;
}


natural operator-(natural* a, natural& b)
{
natural c;
c.resize(a.length());
```

```
// se presupune a >= b (rezultatul este un numar natural)
assert(a.length() >= b.length());
int impumut = 0;
int digit, i;
for(i=0; i<a.length(); i++)
{
digit=a[i]-imprumut;
imprumut=0;
if(i<b.length()) digit-=b[i];
if(digit>=0) c[i]=digit;
else{
imprumut=1;
c[i]=digit+10;
}
}
// se elimina posibilele zerouri din fata numarului
while(c[c.length()-1] == 0)
c.resize(c.length()-1);


return c;
}




int operator<(natural& a, natural& b)
{
int i;
```

```
if(a.length() != b.length()) return a.lentgh() < b.length();
for(i=a.length(); i>=0 && a[i]==b[i]; i--)
;
if(i<0) return TRUE; // sunt egale
else return a[i]<b[i];
}


// clasa intreg



class intreg: public natural {
protected:
int semn;
public:
intreg();
intreg(char*);
intreg(intreg& x): natural(x) { semn = x.get_semn(); }
intreg(natural& x): natural(x) {semn=0;}
intreg(natural& x, int s): natural(x) {semn=s;}
intreg& operator=(intreg&);
~intreg(){};
int length() const {return n;}
int get_semn() {return semn;}
friend natural complement(const natural& x);
friend intreg operator+(intreg& x, intreg& y);
friend ostream& operator<<(ostream& os, const intreg& x);
```

```
};

intreg::intreg() : natural() {semn=1;}

intreg::intreg(char* sir)
{
semn=0;
if(sir[0]=='+')
{
semn=0;
sir++;
}
else
if(sir[0]=='-')
{
semn=1;
sir++;
}
n=strlen(sir);
data=new char[n];
for(int i=0; i<n; i++)
data[i]=sir[n-i-1]-'0';
}
intreg intreg::operator=(intreg& x)
{
if(this != &x)
```

```
{
semn = x.semn;
resize(x.length());
memcpy(data, x.data,n);
}
return *this;
}


natural complement(natural& x)
{
natural z(x);
int i;
for(i=0; i<x.length() && x[i]==0; i++)
z[i++]=10-x[i];
for( ; i<x.length(); i++)
z[i]=9-x[i];
return z;
}


intreg operator+(intreg& x, intreg& y)
{
intreg z;
natural nx(x), ny(y), nz;
int t;
int ss = x.get_semn() + y.get_semn();
switch(ss)
```

```
{
case 0:
return intreg((natural) x + (natural) y);
case 1:
if(ny < nx)
if(x.get_semn()) return intreg(nx-ny,1);
else return intreg (nx-ny,0);
else
if(y.get_semn()) return intreg (ny-nx,1);
else return intreg(ny-nx,0);
case 2:
return intreg ((natural) x + (natural) y,1); // ?!?!
}
}


ostream* operator<<(ostream* out, intreg* a)
{
int i;
if(a.get_semn()==1) out<<"-";
if(a.length()==0) out<<"0";
else
{
// se sare peste eventualele zerouri din fata numarului
for(i=a.length()-1; a[i]==0 && i>0; i--)
;
// se afiseaza numarul
for( ; i>=0; i--)
```

```
out<< ((int) a[i]);
}
return out;
}



int main(void)
{
intreg a,b,c("1");
a = intreg("-110");
b = intreg("-19");
c = a + b + c;
cout<<" c = "<<c<<endl;
return 0;
}
```

Consideram clasa **numar** care este utilizata pentru a lucra cu numere foarte mari. Vom utiliza bazele de numeratie 2, 10 si 16.

```
#include<iostream.h>
#include<string.h>
#include<ctype.h>
#include<assert.h>
```

```
#include<conio.h>


char hexa[] = "0123456789ABCDEF";


/*
```

Numarul lung este pastrat ca un sir de caractere, alocat dinamic, incepand cu cifra cea mai semnificativa in pozitia 0 si incluzand terminatorul de sir.

Functia privata int cifra(char); ne permite sa aflam valoarea unui caracter cifra, conversia inversa facandu-se prin intermediul tabloului hexa[].

Functia resize() realoca spatiul de memorie alocat numarului lung. Daca se aloca mai putina memorie, nu se face realocare ci se trunchiaza numarul in pozitiile cele mai semnificative. Daca se aloca mai multa memorie, in spatiul nou alocat, numarul este alinit la dreapta si completat la stanga cu caractere '0'.

```
*/


class numar{
protected:
int n; // lungime
int b; // baza
char *data; // sir de carac. ce pastreaza numarul si terminatorul
int cifra(char);
public:
numar(int b1=10, char* data=""); // cosntructor de initializare
numar(numar &x; // constructor de copiere
~numar(); // destructor
```

```
numar & operator=(numar& x)/ // operator de atribuire
void resize(int n1);
int size() const { return n; }
int base() const { return b; }
char& operator [] (int i);
//aduna intregul p la numarul lung
numar operator+(int p);
//inmulteste numarul lung cu intregul p
numar operator*(int p);
//imparte numarul lung la intregul p
numar operator/(int p);
//restul impartirii numarului lung la intregul p
int operator%(int p);
//conversie din baza 10 in baza p
numar cv10B(int p);
// conversie din baza 10 in baza b
numar cvB10();
// conversie din baza b in baza p
numar operator> (int p);
friend ostream& operator<<(ostream& os, numar& x);
};
numar :: numar(int b1, char* data1)
{
int n1=strlen(data1);
data = new char [n1+1];
strcpy(data, data1);
n = n1;
b = b1;
```

```
}

numar::numar(numar &x)
{
int n1 = x.size();
data = new char[n1+1];
n = n1;
b = x.base();
for(int i=0;i<=n;i++)
data[i] = x[i];
}

numar::~numar()
{
n = 0;
delete [] data;
}

numar & numar::operator=(numar& x)
{
if(this == &x) return *this; //evita autoatribuirea
int n1 = x.size();
delete [] data;

data = new char [n1+1];
```

```
n = n1;
b = x.base();
for(int i=0; i<=n ;i++)
data[i] = x[i];
return *this;
}


char & numar::operator[] (int i)
{
assert(i>0 && i<= n);
return data[i];
}


void numar::resize(int i)
{
if(n1<n)
memmove(data, data+n-n1);
else
{
char * data1 = new char[n1+1];
memset(data1, '0', n1 - n);
strncpy(data1 + n1 - n, data , n);
data1[n1] = '\0';
delete [] data;
data = data1;
}
```

```
n = n1;
}


int numar::cifra(char c)
{
assert(isdigit(c) || isxdigit(c));
for(int i=0; i<16 && c!=hexa[i]; i++)
;
return i;
}


numar numar::operator+(int p)
{
numar z;
int lz = size();
int sum = p, tr = 0; // suma pe rang si transport
z.resize();
for(int j=lz-1; j>=0; j--)
{
sum += tr + cifra(data[j]);
tr = sum / b;
sum %= b;
z[j]= hexa[sum];
sum = 0;
};
if(tr)
```

```
{
z.resize(++lz);
z[0]=hexa[tr];
}
return z;
};


numar numar::operator*(int p)
{
numar z;
int lz = size();
int prod, tr = 0; // produs partial si transport
z.resize();
for(int j=lz-1; j>=0; j--)
{
prod = tr + cifra(data[j]) * p;
tr = prod / b;
prod %= b;
z[j] = hexa[prod];
};
if(tr)
{
z.resize(++lz);
z[0] = hexa[tr];
}
return z;
};
```

```
numar numar::operator/(int p)
{
numar z;
int lz = size();
int rp; // rest partial
z.resize(lz);
for(int j=0; j < lz; j++)
{
rp = b * rp + cifra(data[j]); // noul rest partial
z[j] = hexa[rp / p];
rp %= p;
};
while(z[0] == '0' && z.size() > 0)
z.resize(--lz);
return z;
};
```

```
numar numar::operator%(int p)
{
numar z;
int lz = size();
int rp; // rest partial
```

```
z.resize(lz);
for(int j=0; j < lz; j++)
{
rp = b * rp + cifra(data[j]); // noul rest partial
z[j] = hexa[rp / p];
rp %= p;
};
while(z[0] == '0' && z.size() > 0)
z.resize(--lz);
return z;
};


numar numar::cvB10()
{
if(b==10) return *this;
numar z(10), y;
int lz = size();
z.resize(2*lz); //convertind in baza 16 avem mai multe cifre
for(int j=0; j<lz; j++)
{
y = z * b;
z = y + cifra(data[j]);
}
while(z[0] == '0' && z.size() > 0)
z.resize(--lz);
return z;
};
```

```
numar numar::cv10B(int p)
{
if(p==10) return *this;
numar z, w(p,"0");
int lz = size();
int j,k,i;
z.resize(lz); //numarul in baza 10
w.resize(4*lz); // numarul in baza p
for(j=0; j<lz; j++) // copiaza obiectul in z
z[j]=data[j];
for(j=0; j<4 * lz; j++) // cifrele se obtin inversate
{
w[j] = hexa[z %p];
z = z / p;
}
for(k=0, i = 4*lz-1; k<i; k++, i--)
{
char c = w[k];
w[k] = w[i];
w[i] = c;
}
j = 0;
while(w[j] == '0')
j++;
w.resize(4*lz - j);
return w;
```

```
};


numar numar:operator>(int p)
{
if(b == p) return *this;
numar z;
z = cvB10();
return z.cv10B(p);
}


ostream* operator<<(ostream& os, numar& x)
{
for(int j=0; j < x.size(); j++)
os <<x[j];
os << endl;
return os;
}


void main(void)
{
clrscr();
numar x(2,"1101001"), y(8,"253647"), z;
cout << x;
cout << y;
z = x.cvB10();
cout << z;
```

```
z = x > 3; cout << z; getch();
}
```

Consideram clasa **polinom** care se utilizeaza pentru lucrul cu polinoame.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
#include<math.h>
#include<iomanip.h>
#include<fstream.h>
#include<stdlib.h>
#include<assert.h>
#define EPS 1.E-3

class polinom{
protected:
int n;
float *data;
public:
polinom(int n1, float *data1);
```

```
    polinom(int n1=0);
    polinom(polinom &p);
    ~polinom(){ if(data) delete[] data; };
    void setn(int m){ n=m; };
    void size() const { return n; };
    void normalize();
    float &operator[] (int i) const;
    polinom &operator+= (polinom &p);
    friend int operator> (const polinom &p, float eps);
    friend polinom operator+ (polinom &p1, polinom &p2);
    friend polinom operator/ (const polinom &p1, const polinom &p2);
    friend polinom operator% (const polinom &p1, const polinom
&p2);
    friend polinom cmmdc (const polinom &p1, const polinom &p2);
    friend ostream &operator<<(ostream &os, const polinom &p);
    friend istream &operator>>(ostream &is, polinom &p);
    };


    polinom::polinom(int n1)
    {
    n=n1;
    data = new float[ n + 1 ];
    for(int i=0;i<=n;i++) data[i]=0;
    }


    polinom::polinom(int n1, float *data1)
    {
```

```
n=n1;
data = new float [ n + 1 ];
memcpy(data, data1, (n+1) * sizeof(float));
}


polinom::polinom(polinom &p))
{
int j;
if(data) delete [] data;
n = p.size();
data = new float[ n + 1 ];
for(j=0; j<=n; j++ )
data[j]=p[j];
}
polinom &polinom::operator=~polinom(const polinom &p))
{
if(this == &p) return *this;
delete [] data;
n = p.size();
data = new float[ n + 1 ];
for(int j=0; j<=n; j++ )
data[j]=p[j];
return *this;
}
void polinom::resize(int n1)
{
if(n1 < = n)
```

```
{
for(int j=n1+1; j<=n; j++) data[j]=0.;
n = n1;
}
else
{
float *data1 = new float[n1+1];
memcpy(data1, data, (n+1) * sizeof(float));
for(int i=n+1; i<=n1; i++) data1[i]=0.;
n = n1;


if(data1) delete[] data;
data = data1;
}
}
void polinom::normalize()
{
int i;
i = n;
while(i>=0 && fabs(data[i]) < EPS) i--;
n = i;
resize(n);
}


float &polinom::operator[] (int i) const
```

```
{
assert(i>=0 && i<=size());
return data[i];
}


int operator> (const polinom &p, float eps)
{
int gr;
gr = p.size();
for(int i=gr; i>=0;i --)
if(data[i] > eps) return 1;
return 0;
}


polinom &polinom ::operator+=(polinom &p)
{
int max;
max = (n > size()) ? n : p.size();
if(max > n ) resize(max);
else p.resize(max);
for(int i=0;i <= max; i++) data[i] += p[i];
normalize();
return *this;
}


polinom operator+ (polinom &p1, polinom &p2)
```

```
{
polinom p(p1);
p += p2;
return p;
}


polinom operator/ (const polinom &p1, const polinom &p2)
{
int n1, n2, q;
n1 = p1.size();
n2 = p2.size();
q = n1 - n2;
polinom a, b;
a = p1;
b = p2;
polinom c(q);
iny i,j;
for(i=q;i>=0;i--)
{
c[i] = a[i+n2] / b[n2];
for(j=n2;j>=0;j--)
a[i+j] -= c[i] * b[j];
}
a.normalize();
b.normalize();
return c;
}
```

```
polinom operator% (const polinom &p1, const polinom &p2)
{
int n1, n2, q;
n1 = p1.size();
n2 = p2.size();
q = n1 - n2;
polinom a, b;
a = p1;
b = p2;
polinom c(q);
iny i,j;
for(i=q;i>=0;i--)
{
c[i] = a[i+n2] / b[n2];
for(j=n2;j>=0;j--)
a[i+j] = a[i+j] - c[i] * b[j];
}
a.normalize();
return a;
}


polinom cmmdc(const polinom &p1, const polinom &p2)
{
polinom a,b;
a = p1;
```

```
b = p2;
int m;
m = p2.size() - 1;
polinom r(m);
do{
r = a % b;
r.normalize();
a = b;
a.normalize();
b = r;
}while(r.size() >= 0 && r > EPS);
return a;
}


ostream & operator<< (ostream &os, const polinom &p)
{


int m = p.size();
for(int j = m; j >= 0; j--)
{
if(p[j] > 0 && j < m) os << "+";
os << p[j]<<"*x^"<<j<<" ";
};
os<<endl;
return os;
}
```

```
istream & operator>> (istream &is, polinom &p)
{
int m;
is>>m;
p.setn(m);
for(int j=0;j<=m;j++)
is>>p[j];
return is;
}


void main ()
{
clrscr();
polinom aa(3);
ifstream f("DATE.IN");
f >> aa;
cout<<"a="<<aa;
polinom b(2);
f >> bb;
cout<<"b="<<bb;
f.close();
polinom c(3);
c = cmmdc(aa, bb);
cout<<" c = "<<c;
getch();
```

}

Proiectati si implementati clasa ***rational*** care sa permita lucrul cu numere rationale.

Constructorul clasei va avea doua argumente: numaratorul, respectiv numitorul rational.

Constructorul poate avea si un singur argument, caz in care al doilea ia implicit valoarea 1, sau nici un argument, caz in care se iau valorile implicite 0 si 1.

Se va sigura un constructor de copiere

Se vor prevedea functii membre pentru:

- accesul la numaratorul, respectiv numitorul numarului rational

- redefinirea operatorilor +=, -=, *=, /= pentru adunarea, scaderea, inmultirea si impartirea numarului rational cu un alt numar rational dat ca argument

Se vor construi urmatoarele *functii prietene*:

- prin redefinire operatorul >> pentru citirea unui numar rational de tastatura

- prin redefinire operatorul << pentru afisarea unui numar rational

- redefinirea operatorului == pentru testul de egalitate a doua numere rationale

- redefinirea operatorilor +, -, *, /, pentru a permite operatii cu doua argumente numere rationale

- redefinirea operatorului de atribuire

Scrieti o functie ***main()*** care citeste n fractii rationale si calculeaza suma lor.

Solutie:

```
#include <iostream.h>
#include <conio.h>
#include <math.h>
class rational
{
// date membru protejate(private)
int ns;//numarator
int nj;//numitor
// functii membru neprotejate
public:
int cmmdc(int a,int b);
// constructor folosit la initializare
rational(int n1=0,int n2=0)
{
ns=n1;nj=n2;
}
// constructor de copiere
rational(rational& r)
{
ns=r.sus();
nj=r.jos();
}
int sus() const { return ns; }
```

```
int jos() const { return nj; }
void setsus(int x){ ns=x;}
void setjos(int y){ nj=y;}
rational& operator+= (const rational& r);
rational& operator-= (const rational& r);
rational& operator*= (const rational& r);
rational& operator/= (const rational& r);
friend istream& operator>>(istream& is, rational& r);
friend istream& operator<<(istream& is, rational& r);
friend rational operator== (const rational& r1, const rational& r2);
friend rational operator+ (const rational& r1, const rational& r2);
friend rational operator- (const rational& r1, const rational& r2);
friend rational operator* (const rational& r1, const rational& r2);
friend rational operator/ (const rational& r1, const rational& r2);
friend void simplifica(rational& r);
};
rational& rational::operator+=(const rational& r)
{
int z;
z = ns * r.jos() + nj * r.sus();
nj= nj * r.jos();
ns= z;
return *this;


}
rational& rational::operator-=(const rational& r)
{
```

```
int t=ns*r.jos()-nj*r.sus();
nj=nj*r.jos();
ns=t;
return *this;
}
rational& rational::operator*=(const rational& r)
{
ns*=r.sus();
nj*=r.jos();
return *this;
}


rational& rational::operator/=(const rational& r)
{
ns/=r.sus();//////////???????ns*=r.sus();
nj/=r.jos();
return *this;
}
int rational::cmmdc(int a, int b)
{
int r;
do{
r=a%b;
a=b;
b=r;
}while(r);
return a;
```

```
}
istream& operator>>(istream& is, rational& r)
{
int x,y;
is>>x>>y;
r.setsus(x);
r.setjos(y);
return is;
}
ostream& operator<<(ostream& os, const rational& r)
{
os<<r.sus()<<" / "<<r.jos();
return os;
}
int operator==(rational& r1, rational& r2)
{
simplifica(r1);
simplifica(r2);
return r1.sus() == r2.sus() && r1.jos() == r2.jos();
}
rational operator+ (const rational& r1, const rational& r2)
{
rational r(r1);
r+=r2;
return r;
}
rational operator- (const rational& r1, const rational& r2)
{
```

```
rational r(r1);
r-=r2;
return r;
}
rational operator* (const rational& r1, const rational& r2)
{
rational r(r1);
r*=r2;
return r;
}
rational operator/ (const rational& r1, const rational& r2)
{
rational r(r1);
r/=r2;
return r;
}


void simplifica(rational& r)
{
int s=r.sus();
int j=r.jos();
int c=r.cmmdc(s,j);
s/=c;
j/=c;
r.setsus(s);
r.setjos(j);
}
```

```
void main(void)
{
clrscr();
rational suma,t;
int n;
// numarul de numere rationale
cout<<"numar de termeni = ";
cin>>n;
cout<<"Introduceti fractiile cate una pe linie"<<endl;
//initial suma este egala doar cu primul nr. rational
cin>>t;
simplifica(t);
suma=t;
//citim si prelucram urmatoarele n-1 nr. rationale
for(int i=1;i<n;i++)
{
cin>>t;
simplifica(t);
suma+=t;
simplifica(suma);
}
cout<<suma<<endl;
getch();
}
```

Sa se definesca o clasa ***punct*** care sa se utilizeze la instantierea punctelor din plan prin coordonate (abcisa si ordonata).

Sa se scrie un program care genereaza obiecte de tip ***punct*** ale caror coordinate se genereaza aleator. Programul afiseza punctele care apartin ecranului, iar in final se indica:

- numarul m al punctelor afisate

- numarul n al punctelor generate

- raportul m/n

Observatie: Punctele care apartin ecranului se incadreaza in dimensiunile acestuia - 80 coloane si 25 de linii.

Solutie:

```
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
class punct
{
// date membru protejate(private)
double x;//abcisa
double y;//ordonata
// functii membru neprotejate
public:
```

```
    punct(double abs=0, double ord=0);
    punct(const punct&);
    //citeste coord. punctului
    void citpunct();
    //afiseaza coordonatele punctului
    void afispunct();
    //translatie pe directia abcisei
    void xtrans(double dx);
    //translatie pe directia ordonatei
    void ytrans(double dy);
    //returneaza abcisa
    double retx();
    //returneaza ordonata
    double rety();
    };
    // constructor pentru instantierea obiectelor de tip punct; implict se
intantiaza originea axelor
    punct::punct(double abs,double ord)
    {
    x=abs; y=ord;
    }
    punct::punct(const punct& p)
    {
    x=p.x; y=p.y;
    }
    void punct::citpunct() // citeste coord. punctului
    {
    cout<<"Dati abcisa ";
```

```
cin>>x;
cout<<"Dati cordonata ";
cin>>y;
}
void punct::afispunct()
{
cout<<" ("<<x<<","<<y<<") ";
}
void punct::xtrans(double dx)
{
x+=dx;
}
void punct::ytrans(double dy)
{
y+=dy;
}
double punct::retx()
{
return x;
}
double punct::rety()
{
return y;
}
void main(void)
{
int m=0,n=0;
clrscr();
```

```
cout<<endl<<"Incepe generarea numerelor "<<endl;
for(;;)
{
double xaleator=rand()%100;
double yaleator=rand()%100;
punct pct(xaleator,yaleator);
n++;
if(xaleator>0 && xaleator<=80 && yaleator>0 &&yaleator<=25)
//afiseaza punctul
{
m++;
pct.afispunct();
if(m%22==0)
{
cout<<endl<<"Apasati o tasta pentru a continua ";
cout<<endl<<"Apasati zero pentru a termina "<<endl;
if(getch()=='0') break;
}
}
}
cout<<"Numarul punctelor afisate "<<m<<endl;
cout<<"Numarul punctelor generate "<<n<<endl;
cout<<"Raportul m/n "<<(double)m/n<<endl;
getch();
}
```

Sa se definesca o clasa ***sir*** care sa se utilizeze la instantierea sirurilor de caractere.

Sa se scrie un program care realizeaza urmatoarele operatii asupra obiectelor de tip ***sir***:

- initializare
- citire de siruri de caractere de la tastatura
- copiere de obiecte de tip ***sir***
- atribuiri de obiecte de tip ***sir***
- afisari de siruri de caractere din compunerea obiectelor de tip ***sir***.

Solutie:

```
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include <string.h>
enum Boolean {false,true};
class sir
{
// date membru protejate(private)
char *psir; //pointer catre zona de memorie in care
// se pastreaza caracterele sirului
int lung; //lungimea sirului(numarul de caractere)
```

```
// functii membru neprotejate
public:
//constructor pt. intializarea obiectului cu pointerul
// spre sirul de caractere; acesta se pastreaza in memoria heap
sir(char *s);
//constructor care rezerva zona de memorie in memoria
//heap si pastreaza sirul vid
sir(int nrcar=70);
//constructor de copiere
sir(const sir&);
//destructor
~sir();
//returneaza lungimea sirului
int retlung();
//afiseaza sirul de caractere
void afsir();
//citeste un sir de caractere
int citsir();
// transfera sirul spre care pointeaza s in zona rezervata pentru
// sirul obiectului current
// daca nu exista zona suficienta, se truncheaza sirul spre care // pointeaza s si se retureaza valorea false; in caz contrar se
// returneaza true



Boolean atribsir(sir *s);
};
```

```
sir::sir(char *s)
{
lung=strlen(s);
psir=new char[lung+1];
strcpy(psir,s);
}
sir::sir(int dim)
{
lung=dim;
psir=new char[lung+1];
*psir='\0';
}
sir::sir(const sir& s)
{
lung=s.lung;
psir=new char[lung+1];
strcpy(psir,s.psir);
}
sir::~sir()
{
delete psir;
}
int sir::retlung()
{
return lung;
}
void sir::afsir()
{
```

```
cout<<psir<<endl;
}
// citeste un sir de caractere de la tastatura si-l pastreaza in
// zona heap
// rezervata pentru obiectul curent
// returneaza 0 la sfarsit de fisier
// -1 la trunchierea sirului citit
// 1, altfel
int sir::citsir()
{
char temp[255];
if(gets(temp)==0) return 0;
strncpy(psir,temp,lung);
*(psir+lung)='\0';
if(strlen(temp) > lung ) return 1;
else return 1;
}
Boolean sir::atribsir(sir* s)
{
strncpy(psir,s->psir,lung);
if(strlen(s->psir) > lung ) return false;
else return true;
}


void main(void)
{
clrscr();
```

```
sir sir1=("Limbajul C++ este un C mai bun");
sir sir2=("Limbajul C++ suporta stilul de programare:\n\
- prin abstractizarea datelor;\n\
- orientata spre obiecte");
sir sir3; // instantiere fara initializare;sir3 contine sirul vid
sir sir4=sir1;
//instantiere folosind constructorul de copiere
//afisarea obiectelor instantiate
cout<<"sir1 : ";sir1.afsir();
cout<<"sir2 : ";sir2.afsir();
cout<<"sir3 : ";sir3.afsir();
cout<<"sir4 : ";sir4.afsir();



//atribuiri de siruri
cout<<"========== Trunchieri ========="<<endl;
if(sir3.atribsir(&sir1) == false)
cout<<"trunchiere la atribuire"<<endl;
sir3.afsir();
if(sir3.atribsir(&sir2) == false)
cout<<"truchiere la atribuire"<<endl;
sir3.afsir();
getch();
}
```

Lista de tip stiva

Sa se realizeze o implementare **template** a unei stive. Stiva este o lista în care adaugarea se face la un capat al listei si stergerea la acelasi capat.

```
#include <iostream.h>
#include <ctype.h>


// Definirea clasei template
template <class C>
class stiva{
private:
int ind,dim;
C *t;
public:
stiva();
stiva(int);
~stiva();
void adaugare(C);
void eliminare();
void parcurgere();
int plina();
```

```
int goala();
};


// Definirea constructorului implicit de tip stiva
template <class C>
stiva<C>::stiva()
{dim=20;t=new C[dim];ind=-1;
cout<<" S-a apelat primul constructor.\n";
}


// Definirea constructorului de tip stiva cu un parametru
template <class C>
stiva<C>::stiva(int nr)
{ dim=nr;t=new C[dim];ind=-1;
cout<<" S-a apelat al doilea constructor.\n";
}


// Adaugarea unui element
template <class C>
void stiva<C>::adaugare(C el)
{ if(plina())
{cout<<"\n Elementul "<<el<<" nu se poate introduce."
"\n Stiva este plina ! \n ";return;}
t[++ind]=el;
}
```

```
// eliminarea unui element
template <class C>
void stiva<C>::eliminare()
{if(goala())
{cout<<"\n Nu se poate sterge ! Stiva s-a golit !\n";return;}
ind--;
}


// Parcurgerea stivei
template <class C>
void stiva<C>::parcurgere()
{ int i;
if(goala()){cout<<"Nu mai exista nici un element !\n";return;}
for(i=0;i<=ind;i++)cout<<" "<<t[i];
cout<<"\n";
}


// Verifica daca stiva este plina
template <class C>
int stiva<C>::plina()
{if(ind>=dim-1) return 1;
else return 0;
}
```

```
// Verifica daca stiva este goala
template <class C>
int stiva<C>::goala()
{ if(ind==-1) return 1;
else return 0;
}


// Destructorul stivei
template <class C>
stiva<C>::~stiva()
{ delete t;
cout<<"S-a distrus stiva !\n";
}


void main(void)
{ int n; double element;
cout<<"\n Cate elemente doriti sa aiba stiva: ";cin>>n;
stiva <double> s(n);
char c;
while(1)
{cout<<" A: Adaugare\n E: Eliminare\n P: Parcurgere\n"
" G: verificare daca stiva este Goala\n"
" U: verificare daca stiva este plina\n I: Iesire\n";
cin>>c;
switch(tolower(c))
{case 'a':cout<<"Elementul: ";cin>>element;
```

```
s.adaugare(element);break;
case 'e':s.eliminare();break;
case 'p':s.parcurgere();break;
case 'g':if(s.goala())cout<<"\n Stiva este goala !\n";
else cout<<"\n Stiva nu este goala ! \n";break;
case 'u':if(s.plina())cout<<"\n Stiva este plina !\n";
else cout<<"\n Stiva nu este plina ! \n";break;
case 'i':cout<<"\Configuratia finala:\n";s.parcurgere();
return;
}
}
}
```

Lista de tip coada

Coada este o lista în care stergerea se face la un capat al listei zi insertia la celalalt capat. La fel ca în cazul listelor, se pot folosi atât alocari dinamice cât si alocari statice.

Un caz particular de coada îl reprezinta coada dublu inlantuita. Acesta permite introducerea si extragerea nodurilor la ambele capete.

Vom implementa o coada folosind definitii **template** si redefinirea operatorilor.

An cadrul clasei template vom defini urmatoarele date:

- inf: informatia folosita pentru fiecare nod;

- ant, urm: adresa nodului anterior zi cea a nodului urmator.

Pentru clasa de tip coada (am denumit-o **queue**) vom defini metodele:
- constructorul clasei, initializeazl capetele cu NULL;
- introducerea elementelor, redefinind operatorul >>;
- parcurgerea cozii, supraîncarcând operatorul <<;
- pentru adaugarea unui element vom redefini operatorul +=;
- stergerea ultimului element din coada, operatorul - -;
- stergerea întregii cozi;
- destructorul clasei, care apeleaza functia de stergere a cozii.

```
#include<iostream.h>
#include<ctype.h>
template <class x>
struct nodd {x inf;
struct nodd<x> *ant,*urm;
};

template<class x>
class queue{
nodd<x>*cap1,*cap2;
public:
queue(){cap1=cap2=NULL;}
friend istream & operator >>(istream&, queue&);
friend ostream & operator <<(ostream&, queue&);
operator--();
void operator+=(int);
```

```
void stergere();
~queue(){stergere();}
};


void main()
{queue<int> s;
char c; int nr;
cin>>s; //crearea cozii
while(1)
{
cin.ignore(10,'\n');
cout<<"\n +: Insertie la sfarsit \n P: Parcurgere "
"\n -: Stergerea primului element \n S: Stergerea totala"
"\n I: Iesire\n";
cin>>c;if(c=='i')break;
switch(toupper(c))
{case 'P':cout<<s;break; //parcurgere
case '+':cout<<"Noul element: ";cin>>nr;
s+=nr;break;
case '-':--s;break;
case 'S':s.stergere();break;
default:cout<<"\n Optiune incorecta \n";
}
}
cout<<"La sfarsit: "<<s;
}
```

```
// Crearea cozii
template <class x>
istream & operator>>(istream &in,queue<x>&s)
{nodd<x>*p;
int nr;


s.cap1=new nodd<x>;s.cap1->urm=s.cap1->ant=NULL;
cout<<"\nIntroduceti informatiile cozii ('.' pt. terminare): ";
cin>>s.cap1->inf;s.cap2=s.cap1;


while(in>>nr)
{p=new nodd<x>;p->inf=nr;
s.cap2->urm=p;p->ant=s.cap2;
s.cap2=p;
}
s.cap2->urm=NULL;
cin.clear();
return in;
}


// Parcurgerea cozii
template <class x>
ostream &operator << (ostream &out,queue<x> &s)
{nodd<x>*p;
```

```
if(s.cap1==NULL){cout<<"Coada vida!";return out;}


for(p=s.cap1;p!=NULL;p=p->urm)
out<<p->inf<<" ";


return out;
}


// Stergerea in intregime a cozii
template <class x>
void queue<x>::stergere()
{if(cap1==NULL)
{cerr<<"\nCoada vida!\n ";return;}
for(cap1=cap1->urm;cap1;cap1=cap1->urm)
delete cap1->ant;
delete cap2;cap2=NULL;
}


// Adaugarea unui nod la coada
template <class x>
void queue<x>::operator+=(int n)
{nodd<x>*p;
if(cap1==NULL)
{cap1=new nodd<x>;cap1->inf=n;cap1->ant=cap1->urm=NULL;
cap2=cap1;
```

```
}
else
{ p=new nodd<x>; p->inf=n;
p->urm=NULL; p->ant=cap2;
cap2->urm=p; cap2=p;
}
}


// Stergerea primului nod din coada
template <class x>
queue<x>::operator--()
{if(cap1==NULL)
{cerr<<"\n Eroare la stergere: coada vida\n ";return 0;}


x nr=cap1->inf;
if(cap1==cap2) {delete cap1;cap1=cap2=NULL;}
else { cap1=cap1->urm;
delete cap1->ant; cap1->ant=NULL;
}
cout<<"A fost sters: "<<nr;
return nr;
}
```

Exemplul 9:

În cele ce urmează vom da ca exemplu pentru definirea unei clase de tip **template**, utilizarea unui tablou cu tipul elementelor si dimensiunea stabilite dinamic, pe parcursul rulării programului.

Ca operatii sunt definite:

- adăugarea unui element;

- afisarea continutului tabloului.

```
#include<conio.h>
#include<iostream.h>


template<class tip>
class tablou
{
tip* t;
int nr;
public:
tablou (int n=10){ t=new tip[nr=n];nr=0;}
~tablou(){delete[] t;}
tip&operator[](int index)
{return t[index];}
void afiseaza_tablou();
void operator+=(tip);
};
```

```
template<class tip> void tablou<tip>::afiseaza_tablou()
{
for(int i=0;i<nr;i++)
{
if(sizeof(t[0])>1) cout<<" ";
cout<<t[i];
}
if(sizeof(tip)>1) cout<<'\n';
}


template<class tip> void tablou<tip>::operator+=(tip a)
{ t[nr++]=a; }


void main()
{
clrscr();
tablou<float>a1(2);
a1+=3.14;
a1+=2.78;
a1.afiseaza_tablou();
tablou<char>a2(20);
a2+='a';
a2+='b';
a2+='c';
a2.afiseaza_tablou();
}
```

Rezultatul executiei este urmatorul:

3.14 2.78

abc

Definirea unei clase template de tipul **multime**. Se vor implementa operatiile de reuniune, intersectie si diferentă între două multimi.

```
#include<iostream.h>


template <class T>
class set {
T * tablou;
public: unsigned int max, nr;
set(unsigned n);
void afisare();
unsigned char este(T &ce);
void include(T &ce);
set operator=(set &b);
```

```
T& operator[](unsigned poz){return tablou[poz];}
friend set operator-(set&, set&);
friend set operator*(set&, set&);
friend set operator+(set&, set&);
};


template<class T> void set<T>:: afisare()
{
cout<<"Multimea este: ";
for(int i=0;i<nr;i++) cout<<tablou[i]<<' ';
cout<<'\n';
}


template<class T> set<T>:: set(unsigned n) // Constructor
{
nr=0;
tablou=new[max=n];
}


template <class T> unsigned char set<T>:: este(T &ce)
// Apartenenta
{
for(int i=0;i<nr;i++)
if (tablou[i]==ce) return 1;
return 0;
}
```

```
template<class T> void set<T>:: include(T &ce)
// Includere element
{
if (!este(ce))
if (nr>=max) cerr<<"Nu s-a rezervat destul spatiu";
else tablou[nr++]=ce;
}


template<class T> set<T> set<T>:: operator=(set<T> &b)
// Atribuire
{
for(nr=0;nr<b. nr;nr++) tablou[nr]=b[nr];
return * this;
}


template<class T> set<T> operator-(set<T>& a, set<T>& b)
// Diferenta
{
set<T> c(a. max);
for(int i=0;i<a. nr;i++)
if (!b. este(a[i])) c. include(a[i]);
return c;
}
```

```
template<class T> set<T> operator*(set<T>& a, set<T>& b) //Intersectie
{
int i = (a. max>b. max) ? a. max: b. max;
set<T> c(i);
for(i=0;i<a. nr;i++)
if (b. este(a[i])) c. include(a[i]);
return c;
}

template<class T> set<T> operator+(set<T>& a, set<T>& b)
//Reuniune
{
set<T> c(a. max+b. max); c=a;
for(int i=0;i<b. nr;i++) c. include(b[i]);
return c;
}

void main()
{
int i,j;
cout<<"Numarul de elemente: ";
cin>>i;
set<int> a(i);
cout<<"Introduceti elementele: ";
```

```
for(i=0;i<a. max;i++)
{
cin>>j;
a. include(j);
}
cout<<"Numarul de elemente: ";
cin>>i;
set<int> b(i);
cout<<"Introduceti elementele: ";
for(i=0;i<b. max;i++)
{
cin>>j;
b. include(j);
}


set<int> c(100);


c=a+b;
cout<<"Reuniune. ";
c. afisare();
c=a*b;
cout<<"Intersectie. ";
c. afisare();
c=a-b;
cout<<"Diferenta A-B. ";
c. afisare();
```

```
c=b-a;
cout<<"Diferenta B-A. ";
c. afisare();
}
```

Se poate propune ca exercitiu redefinirea operatorului **“^”** pentru diferenta simetrică a două multimi.

Sa se afle ce se afiseaza in urma executiei programului :

```
#include<iostream.h>
#include<conio.h>

class C{
public:
C(int n=0, int v[]);
void set(int i,int val){pi[i]=val;}
friend ostream& operator<<(ostream &o, const C&);
private:
int dim;
int *pi;
```

```
};
C::C(int n, int v[])
{
dim=n;
pi=new int[dim];
for(int j=0; j<dim; j++)
{ pi[j]=v[j]; }
}


ostream& operator<<(ostream &o, const C &m)
{
for(int j=0; j<m.dim; j++)
o<<m.pi[j]<<" ";
return o;
}


void main(void)
{
clrscr();
int a[]={1,2,3}, b[]={10,20};
C x(3,a), y(2,b);
cout<<(x=y)<<endl;
y.set(0,1000);
cout<<x<<endl;
getch();
}
```

Sa se afle ce se afiseaza in urma executiei programului :

```
#include<iostream.h>
#include<conio.h>


class C{
public:
C(int n=0, int v[]);
void set(int i,int val){pi[i]=val;}
friend ostream& operator<<(ostream &o, const C&);
C& operator=(C&);
private:
int dim;
int *pi;
};
C::C(int n, int v[])
{
dim=n;
pi=new int[dim];
```

```
for(int j=0; j<dim; j++)
{ pi[j]=v[j]; }
}


ostream& operator<<(ostream &o, const C &m)
{
for(int j=0; j<m.dim; j++)
o<<m.pi[j]<<" ";
return o;
}


C& C::operator=(C& x)
{
delete[] pi;
dim=x.dim;
pi=new int[dim];
for(int j=0; j<dim; j++)
{ pi[j]=x.pi[j]; }
return *this;
}
void main(void)
{
clrscr();
int a[]={1,2,3}, b[]={10,20};
C x(3,a), y(2,b);
cout<<(x=y)<<endl;
```

```
y.set(0,1000);
cout<<x<<endl;
cout<<(x=x)<<endl;
getch();
}
```

Sa se afle ce se afiseaza in urma executiei programului :

```
#include<iostream.h>
#include<conio.h>

class C{
public:
C(int n=0, int v[]);
void set(int i,int val){pi[i]=val;}
friend ostream& operator<<(ostream &o, const C&);
C& operator=(C&);
private:
int dim;
int *pi;
```

```
};
C::C(int n, int v[])
{
dim=n;
pi=new int[dim];
for(int j=0; j<dim; j++)
{ pi[j]=v[j]; }
}
ostream& operator<<(ostream &o, const C &m)
{
for(int j=0; j<m.dim; j++)
o<<m.pi[j]<<" ";
return o;
}


C& C::operator=(C& x)
{
if(this!=&x)
{
dim=x.dim;
delete[] pi;
pi=new int[dim];
for(int j=0; j<dim; j++)
{ pi[j]=x.pi[j]; }
}
return *this;
}
```

```
void main(void)
{
clrscr();
int a[]={1,2,3}, b[]={10,20};
C x(3,a), y(2,b);
cout<<(x=y)<<endl;
y.set(0,1000);
cout<<x<<endl;
cout<<(x=x)<<endl;
getch();
}
```

Sa se afle ce se afiseaza in urma executiei programului :

```
#include<iostream.h>
#include<conio.h>

class Punct{
public:
Punct(int=0, int=0);
protected:
int x,y;
friend ostream& operator<<(ostream &o, const Punct&);
```

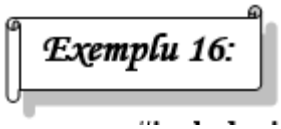
```
};

class Cerc:public Punct{
public:
Cerc(double r=0.0, int x=0, int y=0);
protected:
float raza;
friend ostream& operator<<(ostream &o, const Cerc&);
};

Punct::Punct(int a, int b):x(a),y(b){}

ostream& operator<<(ostream &output, const Punct &p)
{
output<<"Punct"<<'['<<p.x<<", "<<p.y<<']';
return output;
}

Cerc::Cerc(double r, int a, int b):Punct(a,b), raza(r){}

ostream& operator<<(ostream &output, const Cerc &c)
{
output<<"Centru="<<(Punct)(c)<<"; Raza= "<<c.raza;
return output;
```

```
}
```

```
void main(void)
{
clrscr();
Punct *punctPtr=0, p(30,50);
Cerc *cercPtr=0, c(2.7,120,89);
cout<<p<<endl;
cout<<c<<endl;
punctPtr=&c;
cout<<*punctPtr<<endl;
cercPtr=(Cerc *)(punctPtr);
cout<<*cercPtr<<endl;
punctPtr=&p;
cercPtr=(Cerc *)(punctPtr);
cout<<*cercPtr<<endl;
getch();
}
```

Sa se afle ce se afiseaza in urma executiei programului :

```
#include<iostream.h>
#include<conio.h>


class B{
public:
B(){ cout<<"B();"; }
B(B &b) { cout<<"B(B &b); "; }
};
class D:public B{
public:
D(){ cout<<"D();"; }
};


void main(void)
{
clrscr();
B b;
B b1(b);
D d;
D d1(d);
B bd(d);
getch();
}
```



Sa se afle ce se afiseaza in urma executiei programului :

```
#include<iostream.h>
#include<conio.h>


class B{
public:
B(){ cout<<"B(); "; }
~B() { cout<<"~B(); "; }
};
class D:public B{
public:
D(){ cout<<"D(); "; }
~D(){ cout<<"~D(); "; }
};


void main(void)
{
clrscr();
B *b=new B();
```

```
delete b;
b=new D();
delete b;
getch();
}
```

Sa se afle ce se afiseaza in urma executiei programului :

```
#include<iostream.h>
#include<conio.h>

class B{
public:
B(){ cout<<"B(); "; }
~B() { cout<<"~B(); "; }
};
class D:public B{
public:
D(){ cout<<"D(); "; }
~D(){ cout<<"~D(); "; }
};
```

```
void main(void)
{
clrscr();
B *b=new B();
delete b;
b=new D();
delete b;
getch();
}
```

Sa se afle ce se afiseaza in urma executiei programului :

```
#include<iostream.h>
#include<conio.h>
#include<string.h>

class Person{
public:
Person(char *p)
{
```

```
name=new char[strlen(p)+1];
strcpy(name,p);
}
~Person(){ delete[] name; }
private:
char *name;
};


void f()
{
Person *p=new Person("Balanescu");
delete p;
}


void main(void)
{
clrscr();
while(1) f();
getch();
}
```

Exemplu 19:

Sa se afle ce se afiseaza in urma executiei programului :

```
#include<iostream.h>
#include<conio.h>

class Persoana{
public:
Persoana(char *n) { nume=n; }
void afisare(){ cout<<nume<<endl; }
char *nume;
};

Persoana q="Balanescu";
void fvalue(Persoana p){ p.afisare(); }
void faddress(Persoana *p){ p->afisare(); }
void freference(Persoana &p){ p.afisare(); }

void main(void)
{
clrscr();
fvalue(q);
faddress(&q);
freference(q);
```

```
fvalue("Tudor");
freference("Tudor");
getch();
}
```

Sa se afle ce se afiseaza in urma executiei programului :

```
#include<iostream.h>
#include<conio.h>

class B{
public:
B(int i):i(i) { }
protected:
int i;
};

class D1:public B{
public:
D1(int i):B(i){}
void inc() { i++; }
```

```
};


class D2:public B{
public:
D2(int i):B(i){}
void afisare() { cout<<"i="<<i<<endl; }
};


class D: public D1,public D2{
public:
D(int i):D1(i),D2(i){}
};
void main(void)
{
clrscr();
D d(0);
d.inc();
d.afisare();
getch();
}
```

Sa se afle ce se afiseaza in urma executiei programului :

```
#include<iostream.h>
#include<conio.h>


class B{
public:
B(int i=-1):valB(i) {}
B& operator=(const B &b)
{
valB=b.valB;
cout<<"B::op= ";
return *this;
}
private:
int valB;
};


class D: public B{
public:
D(int i, int j):B(i),valD(j){}
D& operator=(const D &d)
{
valD=d.valD;
cout<<"D::op= ";
return *this;
```

```
}
private:
int valD;
};


void main(void)
{
clrscr();
B b(0), b1(1);
getch();
}
```

Sa se afle ce se afiseaza in urma executiei programului :

```
#include<iostream.h>
#include<conio.h>


class A{
int x;
public:
A(int x):x:x(x) {}
```

```
};

class B{
protected:
int x;
public:
B(int x):x(x){}
};

class D:public B{
A a;
B b; // 1
B b(1); // 2
};

void main(void)
{
clrscr();
B b(0), b1(1);
B *pB;
D d(0,0), d1(1,1);
D *pD;
b=b1;
d=d1;
b=d;
pB=&d;
```

```
*pB=d;
pB->operator=(d);
getch();
}
```

Sa se afle ce se afiseaza in urma executiei programului :

```
#include<iostream.h>
#include<conio.h>

class A{
int x;
public:
A(int x):x(x) {}
};

class B{
protected:
int x;
public:
B(int x):x(x){}
```

```
};


class D:public B{
A a;
B b; // 1
public:
void m()
{
x=1; // 2
b.x=1; // 3
}
};


void main(void)
{
clrscr();
B b(1); // 4
getch();
}
```

Sa se afle ce se afiseaza in urma executiei programului:

```
#include<iostream.h>
#include<conio.h>


class A{
public:
void m(){ cout<<"A:m()"<<endl; }
virtual void v(){ cout<<"A:v()"<<endl; }
};


class B:public A{
private:
void m(){ cout<<"B:m()"<<endl; }
virtual void v(){ cout<<"B:v()"<<endl; }
};


void main(void)
{
clrscr();
A a, *p;
B b;
a.m();
p=&b;
p->v();
getch();
```

```
}
```

Sa se afle ce se afiseaza in urma executiei programului:

```
#include<iostream.h>
#include<conio.h>

class A{
public:
void s(){ cout<<"void A::s()"<<endl; }
void s(int i){ i++;cout<<"void A::s(int i)"<<endl; }
virtual void v(){ cout<<"A::v()"<<endl; }
virtual void v(int i)
{
i++;
cout<<"virtual void A::v()"<<endl;
}
};

class B:public A{
```

```
public:
// 1
int s()
{
cout<<"int A::s()"<<endl;
return 1;
}
// 2
void s(int i)
{
i++;
cout<<"void A::s(int i)"<<endl;
return 1;
}
// 3
virtual void v(int i)
{
i++;
cout<<"virtual void A::v()"<<endl;
}
// 4
virtual int v()
{
cout<<"int void A::v()"<<endl;
return 1;
}
```

```
};


void main(void)
{
clrscr();
getch();
}
```

Bibliografie

[1.] Brian Kernighan, Dennis Ritchie - The C Programming Language; Editura Prentice Hall 1978, 1985;

[2.] Bjarne Stroustrup - The C++ Language (second edition); Editura Addison Wesley, 1993;

[3.] Margaretz Ellis, Bjarne Stroustrup - The Annotated C++ Reference Manual; Editura Addison Wesley, 1991;

[4.] Borland C++.Version 4.5 - Programmers Guide, Borland International, 1995;

[5.] Kacso Adrian, Kacso Daniela - Turbo C - Tehnici de programare; Editura MicroInformatica, Cluj, 1994;

[6.] Ionut Muslea - Initiere în C++. Programare orientată pe obiecte; Editura MicroInformatica, Cluj, 1992;

[7.] Ionut Muslea - C++ pentru avansati; Editura MicroInformatica, Cluj, 1994;

[8.] Octavian Catrina, Iulian Cojocaru-Turbo C++; Editura Teora, Bucuresti, 1993;

[9.] Cristian Giumale, Eugenia Kalisz, Alexandru Păunoiu, Valentin Cristea - Introducere în limbajul C; Editura Teora, Bucuresti,1992;

[10.] Clara Ionescu, Ioan Zsako - Structuri arborescente cu aplicatiile lor; Editura Tehnică, Bucuresti, 1990;

[11.] Dan Somnea si Doru Turturea - Initiere în C++: Programarea orientată pe obiecte; Editura Tehnică, Bucuresti, 1990;

[12.] Nabajyoti Barakakati - Borland C++ 4; Ghidul programatorului; (traducere) Editura Teora, Bucuresti,1996;

[13.] Herbert Schilldt - C++ manual complet; (traducere), Editura Teora, Bucuresti, 1997;

[14.] Herbert Schilldt - C manual complet; (traducere), Editura Teora, Bucuresti, 1998;

[15.] Mickey Williams - Bazele Visual C++ 4; (traducere), Editura Teora, Bucuresti, 1997;

[16.] Octavian Aspru - Tehnici de programare; Editura Adias, Râmnicu-Vâlcea, 1997;

[17.] Tudor Sorin-Tehnici de programare;(Proiect de manual pentru clasa a X-a); Editura L&S InfoMat, Bucuresti, 1998;

[18.] Tudor Sorin - Bazele programării în C++; (Proiect de manual pentru clasa a XI-a); Editura L&S InfoMat, Bucuresti, 1997;

[19.] Liviu Negrescu - Limbajele C si C++ pentru începători; Editura MicroInformatica, Cluj, 1995;

[20.] Liviu Negrescu - Limbajul C; (Proiect de manual pentru clasa a XI-a); Editura Computer Libris Agora, Cluj, 1997;

[21.] George Daniel Mateescu - C++ limbaj si programe, Editura Petrion, Bucuresti, 1998;

[22.] Bogdan Pătrut - Aplicatii în C/C++; Editura Teora, Bucuresti, 1998;

[23.] F. Munteanu, T. Ionescu, G. Muscă, D.Tătaru, S.M.Dascălu - Programarea calculatoarelor - Editura Didactică si Pedagogică, Bucuresti, 1995;

[24.] Dorian Stoilescu - Culegere de C/C++, (Proiect de manual pentru Aplicatii practice de laborator); Editura Radial, Galati, 1997;

[25.] Mihai Oltean - Programarea jocurilor matematice; Editura MicroInformatica, Cluj, 1995;

[26.] Seria Gazeta de Informatică;

[27.] Dr. Kris Jamsa, Lars Klander – Torul despre C si C++. Manualul fundamental de programare in C si C++ ; Editura Teora, Bucuresti, 1999;

[28.] Octavian Patrascoiu, Gheorghe Marian, Nicolae Mitroi - Elemente de grafuri si combinatorica. Metode, algoritmi si programe, Editura All, Bucuresti, 1994;

[29.] Thomas H. Cormen, Charles E. Leiserson, Ronald R. Rivest – Introducere in algoritmi, Editura Computer Libris Agora, Cluj-Napoca, 2000;

[30.] Mircea Dorin Popovici, Mircea Ioan Popovici – C++. Tehnologia orientata spre obiecte, Editura Teora, Bucuresti, 2002.

[31.] Adrian Runceanu – Metode si tehnici de programare – limbajul C++, Editura Academica Brancusi, Targu-Jiu, 2003.

##########

Despre autor

Adrian Runceanu *(n. 1968) este lector universitar doctor la Universitatea Constantin Brâncuşi din Târgu-Jiu, Facultatea de Inginerie, Departamentul de Automatică, Energie şi Mediu. Este licenţiat în specialitatea Informatică (1995), al Facultăţii de Matematică-Informatică din cadrul Universităţii din Craiova. În anul 2009 a susţinut teza de doctorat cu titul "Algoritmi de creştere a vitezei de*

transfer a informației în bazele de date distribuite", în domeniul Știința calculatoarelor, la Facultatea de Automatică, Calculatoare și Electronică din cadrul Universităţii din Craiova.

Domeniile de cercetare sunt legate de baze de date distribuite, programare web, programarea calculatoarelor și e-learning.

Tine legatura cu Adrian Runceanu

Multumim pentru citirea acestei carti! Urmareste in continuare activitatea acestui autor prin canalele sociale de mai jos.

Facebook: ***http://www.facebook.com/adrianrunceanu***

Research Gate: ***http://www.researchgate.net/profile/Adrian_Runceanu/***

LinkedIn: ***http://ro.linkedin.com/in/adrianrunceanu***

Site: ***http://www.runceanu.ro/adrian/***

Editura Digitala este o platforma online de publicare carti digitale.
Intra si tu pe www.edigitala.ro si publica-ti cartea.
Website: www.edigitala.ro
Email: office@edigitala.ro